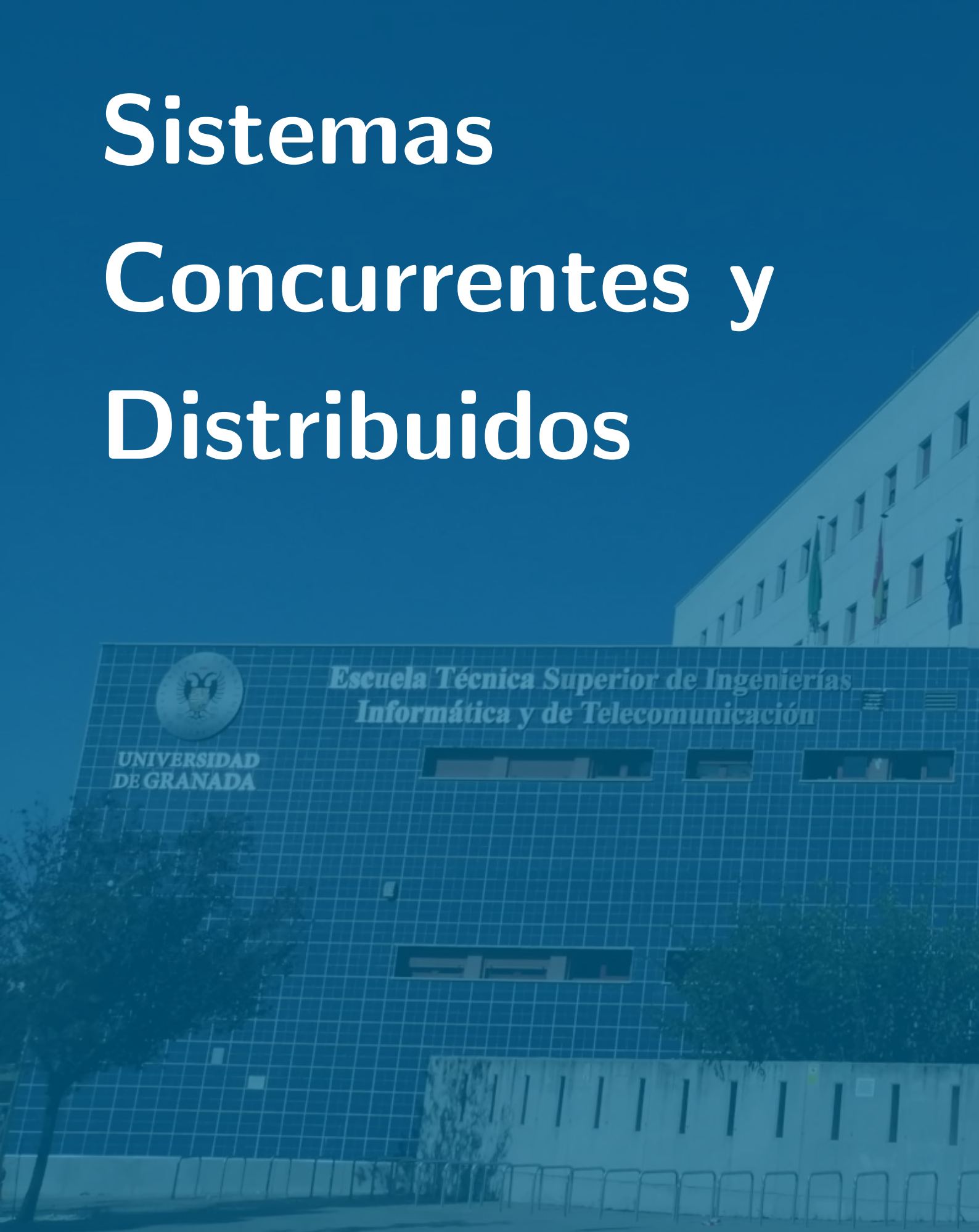
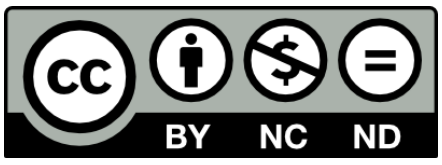
Sistemas Concurrentes y Distribuidos



Los Del DGIIM, losdeldgiim.github.io

Doble Grado en Ingeniería Informática y Matemáticas
Universidad de Granada



Sistemas Concurrentes y Distribuidos

Los Del DGIIM, `losdeldgiim.github.io`

José Juan Urrutia Milán
Arturo Olivares Martos

Granada, 2024-2025

Índice general

1. Introducción a la Programación Concurrente

1.1. Conceptos básicos

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

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

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

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

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

1.1.1. Comparación de programas concurrentes con secuenciales

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

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

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

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

1.1.2. Definición de concurrencia

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

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

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

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

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

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

1.1.3. Axiomas de la programación concurrente

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

Estos axiomas son:

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

2. **Consistencia de los datos tras un acceso concurrente.**
   El entrelazamiento de instrucciones atómicas preserva la consistencia de los resultados de las operaciones. Es decir, si tenemos muchos procesos actuando a la vez sobre un conjunto de datos compartidos, debemos estar seguros de que los accesos a los mismos no los estropeen.

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

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

4. **Independencia de la velocidad de los procesos.**
   No puede hacerse ninguna suposición en la velocidad de ejecución de un proceso, ya que este puede verse suspendido o ralentizado, salvo que esta es positiva.

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

5. **Hipótesis del progreso finito.**

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

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

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

1.2. Modelos para creación de procesos en un programa

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

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

1.2.1. Grafos de Sincronización

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

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

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

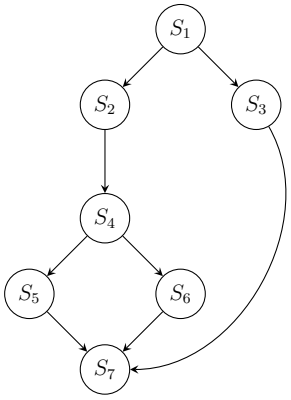


Figura 1.1: Ejemplo de DAG

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

1.2.2. Definición estructurada de procesos

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

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

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

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

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

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

1.2.3. Definición no estructurada de procesos

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

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

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

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

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

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

1.3. Exclusión mutua y sincronización

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

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

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

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

Ejemplo. Por ejemplo, ante el siguiente código concurrente:

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

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

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

- Podría ejecutarse el primer proceso antes que el segundo, dejando la variable `x` a 1 y el segundo le cambiaría el valor a 0.

- El primer proceso puede leer la variable `x` antes de que el segundo escriba en ella, leyendo 0; y podría escribir en ella después de que lo haga el segundo proceso, escribiendo finalmente un 1.

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

Sin embargo, ante el siguiente código concurrente:

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

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

Paradigma del Productor Consumidor

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

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

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

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

1. `L, E, L, E, ...`, porque leemos una lectura de la variable antes de que el productor escriba en ella, leyendo un valor indeterminado y puediendo provocar el fallo del programa.

2. `E, L, E, E, L, ...`, porque el consumidor se ha perdido una escritura del productor en la variable, que puede hacer que cambie la salida del programa a una errónea.

3. `E, L, L, E, L, ...`, porque el consumidor ha usado un mismo dato dos veces, que también puede resultar en un mal funcionamiento del programa.

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

1. El consumidor no puede leer la variable hasta que el productor haya escrito en ella. Cuando el consumidor lee, debe esperar a que el productor proporcione un nuevo dato antes de volver a leer.

2. El productor no puede escribir un nuevo valor hasta que el consumidor haya leido el último dato escrito (salvo en el primer valor a escribir).

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

- El consumidor se detenga la primera vez hasta que el productor escriba en la variable.

- Se impida un segundo ciclo del consumidor hasta que se produzca el siguiente dato.

- Se impida un segundo ciclo del productor hasta que el dato anterior no haya sido leído por el consumidor.

1.4. Propiedades de los sistemas concurrentes

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

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

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

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

1.4.1. Propiedades de seguridad (*safety*)

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

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

Ejemplos de problemas donde vemos propiedades de seguridad son:

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

1.4.2. Propiedades de vivacidad (*liveness*)

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

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

Ejemplos de problemas donde vemos propiedades de vivacidad son:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.5.1. Corrección de los programas concurrentes

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

Diremos que un programa secuencial es *parcialmente correcto* si, supuesto que este termine, entonces los resultados que obtiene tras ejecutarse son esperados.

En un programa secuencial, hay un único conjunto de datos de entrada que provoca un único conjunto de datos de salida. Esto no sucede en programas concurrentes, ya que el indeterminismo en la ejecución provoca distintas trazas posibles

del programa, y es bastante probable que todas las trazas posibles no provoquen los mismos resultados.

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

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

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

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

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

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

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

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

1.5.2. Introducción a la Lógica de Hoare

Axiomática del lenguaje

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

Nuestro SLF estará formado por:

- Símbolos: Como sentencias del lenguaje de programación, variables proposicionales, operadores, ...
- Fórmulas: Secuencias de símbolos bien formadas[2].

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

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

- Axiomas: Proposiciones que mediante un consenso se consideran verdaderas.
- Reglas de inferencia o de derivación: Reglas que nos permiten derivar fórmulas ciertas a partir de axiomas o de fórmulas que ya conocemos que son ciertas.

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

Notación. Notaremos a las reglas de inferencia por:

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

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

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

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

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

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

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

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

Lógica proposicional

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

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

[3]Cuyos elementos interpretamos como verdadero y falso.

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

De esta forma, diremos que:

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

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

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

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

1.5.3. Lógica de programas

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

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

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

Un triple $\{P\}$ S $\{Q\}$ se interpreta como cierto si S es ejecutado en un estado del programa que satisface P y, si la ejecución de S termina, el estado en el que S termina satisface Q, independientemente de los efectos producidos por los entrelazamientos de instrucciones atómicas de S.

Notemos que de esta forma el triple $\{V\}$ `while true do begin end` $\{F\}$ es siempre cierto, ya que S se ejecuta en un estado del programa que satisface P y S nunca termina.

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

$$\{P\}$$

Al conjunto de los estados del programa que verifican P.

De esta forma, $\{V\}$ es el conjunto de todos los estados de un programa, ya que todos los estados de dicho programa verifican V. Análogamente, $\{F\}$ se corresponde con el conjunto vacío.

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

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

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

Observación. Notemos que, dados cualesquiera asertos $\{P\}$ y $\{Q\}$, podemos pensar en:

- $\{P \wedge Q\}$ como en $\{P\} \cap \{Q\}$.
- $\{P \vee Q\}$ como en $\{P\} \cup \{Q\}$.
- $\{P\} \rightarrow \{Q\}$ como en $\{P\} \subseteq \{Q\}$

De esta forma, notemos que siempre se verifica que:

- $\{V \wedge P\} = \{P\}$.
- $\{V \vee P\} = \{V\}$.
- $\{F \wedge P\} = \{F\}$.
- $\{F \vee P\} = \{P\}$.
- $\{F\} \rightarrow \{P\} \rightarrow \{V\}$.
- $\{P \wedge Q\} \rightarrow \{P\} \rightarrow \{P \vee Q\}$.

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

Verificación de programas secuenciales

Enumeramos ahora los axiomas de nuestra Lógica de programas:

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

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

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

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

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

de donde:

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

□

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

Regla de la consecuencia (1).

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

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

Regla de la consecuencia (2).

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

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

Regla de la composición.

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

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

Regla de la conjunción.

$$\frac{\{P_1\}\ S\ \{Q_1\}, \{P_2\}\ S\ \{Q_2\}}{\{P_1 \wedge P_2\}\ S\ \{Q_1 \wedge Q_2\}}$$

Es decir, si tenemos distintas pre y poscondiciones para una misma instrucción, podemos juntarlas todas en conjunción.

Regla del *if*.

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

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

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

es cierta, tendremos que probar que las tripletas

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

son ciertas.

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

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

Donde a la proposición I la llameremos invariante.

1.5.4. Verificación de sentencias concurrentes

Sabemos ya hacer demostraciones para verificar la corrección de programas secuenciales. Será de nuestro interés ahora permitir la ejecución concurrente de dichos flujos de ejecución secuenciales, con el objetivo de probar la corrección de un programa concurrente.

Si entendemos la ejecución de un programa concurrente como un entrelazamiento de las instrucciones atómicas ejecutadas por los procesos del programa, entonces hemos de tener en cuenta para la demostración de corrección que no todas las secuencias de entrelazamiento resultan ser aceptables. Para poder programar correctamente, usamos sentencias de sincronización, tales como:

- Secciones críticas en el código para evitar condiciones de carrera.
- Sincronización con una condición (hacer que un proceso espere hasta que se dé una determinada condición en el estado del programa).

Usando estas operaciones, conseguiremos hacer correctos nuestros programas concurrentes (que es de lo que trata la asignatura), para luego demostrar matemáticamente que, efectivamente, dichos programas son correctos.

Dados varios triples de Hoare que representan secciones de programas secuenciales de los que hemos probado su corrección parcial, tratamos ahora de introducir estas secciones de programa en un programa concurrente.

Sin embargo, puede suceder que un proceso ejecute una instrucción atómica de su región de código que haga falsa la precondición o la poscondición de una sentencia que esté siendo simultáneamente ejecutada por otro proceso, invalidando su corrección.

Ejemplo. Por ejemplo, tenemos los dos siguientes triples:

$$\{x = 0\}\ x = x + 2\ \{x = 2\}$$
$$\{V\}\ x = 0\ \{x = 0\}$$

cuya corrección puede comprobarse fácilmente usando el axioma de asignación.

Ahora, nos disponemos a ejecutar el siguiente código concurrente:

```
1 x = 0;
  cobegin
    <x = x + 2;> || <x = 0;>
  coend
```

donde podemos ver que la ejecución de una instrucción *interfiere* con la poscondición de la otra instrucción:

- La poscondición $\{x = 0\}$ de la instrucción de la derecha no se cumple tras la ejecución de `<x = x + 2;>` (en caso de que esta se ejecute después).
- La poscondición $\{x = 2\}$ de la instrucción de la izquierda no se cumple tras la ejecución de `<x = 0;>` (en caso de que esta se ejecute después).
- Notemos que la ejecución de una instrucción no *interfiere* con la precondición de la otra.

La ejecución de una instrucción hace falsa la poscondición de la otra, invalidando la demostración que hicimos del trozo de programa secuencial y, por tanto, del programa concurrente.

Sin embargo, podemos hacer un cambio en los triples iniciales para no tener este problema: usando la primera regla de la consecuencia, podemos relajar las poscondiciones de ambos triples:

$$\{x = 0\}\ x = x + 2\ \{x = 0 \vee x = 2\}$$
$$\{V\}\ x = 0\ \{x = 0 \vee x = 2\}$$

haciendo que estos sigan siendo correctos. Si ahora tratamos de ejecutar de forma concurrente estas dos instrucciones, observamos que independientemente de la traza de ejecución, la ejecución de una instrucción no hace falsas las pre o poscondición de la otra.

Antes de proceder a la formalización del concepto de interferencia, hemos de comentar ciertos detalles:

- Una acción atómica elemental o sección crítica realiza una transformación indivisible del estado del programa, de forma que cualquier estado intermedio que pudiera existir no sería visible para el resto de los procesos.

  En los programas concurrentes, puede suceder que las asignaciones no sean atomicas, por estar típicamente implementadas por varias instrucciones máquina:

```
1 y = 0; z = 0;
  cobegin
    x = y + z; || y = 1; z = 2
  coend
```

  El programa superior puede dar como resultados esperados de `x` 0, 1, 3 (como el lector habrá podido adivinar) y 2. Este último valor sería resultado de haber leído en la instrucción de la izquierda el valor de `y`, la ejecución de las dos instrucciones de la derecha, y finalmente añadir `z` al valor leído, guardándolo en `x`.

  Se trata de un ejemplo curioso, ya que $z + y = 2$ no se correponde con ningún estado del programa. Esta situación puede resolverse con el uso de secciones críticas, ya que en este caso lo que falla es el programa concurrente, que está mal programado.

- Una expresión que no hace referencia a ninguna variable modificada por otro proceso será evaluada de forma atómica, ya que ninguno de los valores de la variable puede ser modfiicada mientras la expresión resulta evaludada.

  De esta forma, si consideramos el siguiente programa concurrente en el que los procesos se encuentran usando variables disjuntas:

```
x = 0; y = 0;
cobegin x = x + 1; || y = y + 1; coend
```

  La ejecución de una instrucción no tiene nada que ver con la ejecución de la otra.

Interferencia

Dado un triple $\{P\}S\{Q\}$, este puede contener varios asertos: $\{P\}$, $\{Q\}$ y cualquiera que sirve como pre y poscondición entre las acciones atómicas incluidas en la sección de código S.

Ejemplo. De esta forma, el triple

$$\{V\}\ x = 0;\ x = x + 1;\ \{x = 1\}$$

contiene los asertos $\{V\}$, $\{x = 0\}$ y $\{x = 1\}$. Sin embargo, el triple

$$\{V\}\ <x = 0;\ x = x + 1;>\ \{x = 1\}$$

contiene los asertos $\{V\}$ y $\{x = 1\}$, ya que la ejecución de las dos instrucciones se hace de forma atómica y ningún otro proceso puede ver el estado intermedio ($x = 0$) de la variable x, ya que como hemos mencionado en el primer punto, las regiones críticas realizan una transformación **indivisible** del estado del programa.

Definición 1.4 (Interferencia)**.** Dado un triple $\{P\}S\{Q\}$ y una sentencia atómica a[4] con precondición $pre(a)$, decimos que a no interfiere con $\{P\}S\{Q\}$ si para todo aserto A del triple se cumple el triple

$$NI(A, a) :\equiv \{A \wedge pre(a)\}a\{A\}$$

Es decir, si el aserto A es invariante para la sentencia a.

Observación. Notemos que, en la definición anterior, hemos empleado el símbolo $\equiv$ para denotar la igualdad entre dos triples. Este abuso de la notación será no obstante fácil de distinguir en el contexto.

Definición 1.5. Dados n triples $\{P_i\}S_i\{Q_i\}$, decimos que están libres de interferencia si S_i no interfiere con $\{P_j\}S_j\{Q_j\}$, para cada par (i, j) con $i \neq j$.

[4]Esta puede contener tantas líneas de código como queramos, lo importante es que su ejecución se haga de forma atómica.

La falta de interferencia significa que la ejecución atómica de pasos de un proceso del programa nunca falsifica los asertos usados en las demostraciones individuales de los otros procesos.

Esto es de especial relevancia en los programas concurrentes, ya que para demostrar su corrección, primero hemos de demostrar que la ejecución secuencial de cada proceso es correcta (en caso de no serlo, directamente el programa concurrente no es correcto), para luego demostrar que la ejecución de un proceso no interfiere con otro, para así garantizar que, aunque tengamos varios procesos ejecutándose de forma concurrente, como no interfieren entre sí, no se modifica la corrección del programa.

Veremos próximamente que con garantizar la corrección secuencial de los procesos y la no interferencia entre los mismos, nos es suficiente para garantizar la corrección de los programas concurrentes, gracias a este razonamiento intuitivo que acabamos de realizar (lo único que cambia al ejecutar los procesos de forma concurrente es que puedan modificar variables de otros procesos, interfiriendo en su ejecución, sin embargo, si no interfieren en la misma, todo saldrá bien).

Ejemplo. Recuperando el ejemplo del inicio de la sección, nos disponemos a demostrar que los triples $\{x = 0\}\ x = x + 2\ \{x = 0 \vee x = 2\}$, $\{V\}\ x = 0\ \{x = 0 \vee x = 2\}$ están libres de interferencia. Para ello, hemos de demostrar $2 \cdot 2 = 4$ triples (ya que tenemos dos instrucciones, cada una con 2 asertos):

$$
\begin{aligned}
NI(x = 0,\ x = 0) &\equiv \{x = 0 \wedge V\}\ x = 0\ \{x = 0\} \\
NI(x = 0 \vee x = 2,\ x = 0) &\equiv \{(x = 0 \vee x = 2) \wedge V\}\ x = 0\ \{x = 0 \vee x = 2\} \\
NI(V,\ x = x + 2) &\equiv \{V \wedge x = 0\}\ x = x + 2\ \{V\} \\
NI(x = 0 \vee x = 2,\ x = x + 2) &\equiv \{(x = 0 \vee x = 2) \wedge x = 0\}\ x = x + 2\ \{x = 0 \vee x = 2\}
\end{aligned}
$$

1. Para el primer triple:

   $$\{x = 0 \wedge V\}\ x = 0\ \{x = 0\}$$

   Tenemos que $\{x = 0 \wedge V\} \equiv \{x = 0\}$, por lo que basta con probar:

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

   Usando el axioma de asignación:

   $$\{V\} \equiv \{0 = 0\} \equiv \{x = 0\}_0^x\ x = 0\ \{x = 0\}$$

   Y como $\{x = 0\} \rightarrow \{V\}$, usando la segunda regla de consecuencia, tenemos ya probado $NI(x = 0,\ x = 0)$.

2. Para el segundo triple

   $$\{(x = 0 \vee x = 2) \wedge V\}\ x = 0\ \{x = 0 \vee x = 2\}$$

   Tenemos que $\{(x = 0 \vee x = 2) \wedge V\} \equiv \{x = 0 \vee x = 2\}$, por lo que basta con probar:

   $$\{x = 0 \vee x = 2\}\ x = 0\ \{x = 0 \vee x = 2\}$$

Usando el axioma de asignación:

$$\{V\} \equiv \{V \vee F\} \equiv \{0 = 0 \vee 0 = 2\} \equiv \{x = 0 \vee x = 2\}_0^x \; x = 0 \; \{x = 0 \vee x = 2\}$$

Y como $\{x = 0 \vee x = 2\} \rightarrow \{V\}$, tenemos el triple probado usando otra vez la segunda regla de la consecuencia.

3. Para el tercer triple:

$$\{V \wedge x = 0\} \; x = x + 2 \; \{V\}$$

Como $\{V \wedge x = 0\} \equiv \{x = 0\}$, probamos:

$$\{x = 0\} \; x = x + 2 \; \{V\}$$

Dado que el triple $\{x = 0\} \; x = x + 2 \; \{x = 2\}$ es cierto (se comprueba trivialmente con el axioma de asignación) y como $\{x = 2\} \rightarrow \{V\}$, tenemos el triple probado usando la primera regla de la consecuencia.

4. Para el cuarto triple:

$$\{(x = 0 \vee x = 2) \wedge x = 0\} \; x = x + 2 \; \{x = 0 \vee x = 2\}$$

Como:

$$\{(x = 0 \vee x = 2) \wedge x = 0\} \equiv \{x = 0\}$$

Tenemos que probar:

$$\{x = 0\} \; x = x + 2 \; \{x = 0 \vee x = 2\}$$

Usando el axioma de asignación:

$$\{x = 0 \vee x = 2\}_{x+2}^x \; x = x + 2 \; \{x = 0 \vee x = 2\}$$

$$\{x = 0 \vee x = 2\}_{x+2}^x \equiv \{x + 2 = 0 \vee x + 2 = 2\} \equiv \{x = -2 \vee x = 0\}$$

Y como $\{x = 0\} \rightarrow \{x = 0 \vee x = -2\}$, lo tenemos demostrado usando la segunda regla de consecuencia.

A continuación, veremos una regla de inferencia relacionada con lo que explicamos tras la definición de interferencia, y es que nos es suficiente con demostrar la corrección secuencial de los procesos y la no interferencia entre los mismos para poder garantizar la corrección de un programa concurrente.
Un programa concurrente pueden entenderse como la ejecución secuencial de estructuras `cobegin-coend`, por lo que será suficiente con demostrar la corrección parcial de cada una de estas estructuras para luego demostrar la corrección parcial de todo el programa, que se reducirá a una demostración de un programa secuencial.

Regla de la composición concurrente segura de procesos.

$$\frac{\{P_i\} \; S_i \; \{Q_i\} \text{ son triples libres de interferencia } 1 \leqslant i \leqslant n}{\{P_1 \wedge P_2 \wedge \ldots \wedge P_n\} \; \texttt{cobegin} \; S_1 \; || \; S_2 \; || \; \ldots \; || \; S_n \; \texttt{coend} \; \{Q_1 \wedge Q_2 \wedge \ldots \wedge Q_n\}}$$

Ejemplo. Aplicamos ahora la regla de la composición concurrente al ejemplo que venimos manejando. Como hemos visto, los triples $\{x = 0\}\ x = x+2\{x = 0 \vee x = 2\}$ y $\{V\}\ x = 0\ \{x = 0 \vee x = 2\}$ están libres de interferencia, por lo que podemos aplicar la regla de la composición concurrente, llegando a que el triple

$$\{x = 0\}$$
$$\texttt{cobegin}\ \ x = x + 2;\ \ ||\ x = 0;\ \ \texttt{coend}$$
$$\{x = 0 \vee x = 2\}$$

es cierto.

Ejemplo. Ante el siguiente código:

```
1 x = 0; y = 0;
  cobegin x = y + 1; || y = x + 1; coend
```

Notemos que los posibles resultados de ejecución del mismo son (donde se ha notado por $l(x)$ la lectura de la variable x y por $e(x)$ a la escritura en la variable x, respectivamente con y):

Traza	x	y	Traza	x	y	Traza	x	y
$l(x)$	0	0	$l(x)$	0	0	$l(y)$	0	0
$l(y)$	0	0	$e(y)$	0	1	$e(x)$	1	0
$e(x)$	1	0	$l(y)$	0	1	$l(x)$	1	0
$e(y)$	1	1	$e(x)$	2	1	$e(y)$	1	2

Vemos en la tabla que hay tres posibles resultados del programa:

1. Uno en el que primero se leen las variables y luego se modifican.
2. Otro en el que primero se modifica y y luego x.
3. Un último en el que primero se modifica x y luego y.

Vemos que al tener varios procesos que escriben y modifican varias variables, debemos distinguir varias casuísticas para luego determinar los posibles resultados del programa.

Sin embargo, ante el siguiente código:

```
1 x = 0; y = 0;
  z = x;
  cobegin x = y + 1; || y = z + 1; coend
```

obtenemos los siguientes posibles resultados:

Traza	x	y	Traza	x	y	Traza	x	y
$l(z)$	0	0	$l(z)$	0	0	$l(y)$	0	0
$l(y)$	0	0	$e(y)$	0	1	$e(x)$	1	0
$e(x)$	1	0	$l(y)$	0	1	$l(z)$	1	0
$e(y)$	1	1	$e(x)$	2	1	$e(y)$	1	1

Donde sólo hay una única variable compartida que es leida y escrita por un único proceso, el número de casos a tener en cuenta disminuye, ya que sólo debemos distinguir dos casos:

1. Al ejecutar el código de la izquierda, la variable y todavía no ha sido modificada.

2. Al ejecutar el código de la izquierda, la variable y ha sido modificada ya.

Esta propiedad de los programa recibe el nombre de **como máximo una vez**, donde solo encontramos una variable que es leída por varios procesos pero que solo es modificada por uno de ellos. De esta forma, el número de casuísticas disminuye considerablemente, ya que sólo hemos de distinguir el caso de que la variable haya sido modificada y el caso de que todavía no lo haya sido.

1.5.5. Verificación usando invariantes globales

La demostración de verificación de programas concurrentes usando la regla de la composición concurrente es tediosa, ya que es necesario comprobar la veracidad de muchos triples antes de poder aplicarla. Es por tanto que buscamos otra forma predominante de probar los programas concurrentes.

Los invariantes globales (IG) de un programa pueden entenderse como expresiones definidas a partir de las variables globales de un programa. Estas suelen definirse como un predicado de la Lógica de Programas que captura la relación que existe entre las variables compartidas por los procesos de un programa concurrente.

Si cualquier aserto $\{C\}$ de un triple $\{P_i\}\ S_i\ \{Q_i\}$ se puede escribir como una conjunción del tipo $\{IG \wedge L\}$ donde IG es un invariante global del programa y $\{L\}$ es un predicado en el que intervienen solo variables de un proceso o parámetros de una función, entonces las demostraciones de los procesos secuenciales estarán libres de interferencias.

De esta forma, para que un predicado $\{I\}$ definido a partir de las variables compartidas entre los procesos pueda ser considerado un IG válido, se ha de cumplir que:

1. Sea cierto para los valores iniciales de las variables del programa que aparecen en $\{I\}$.

2. Se ha de poder demostrar la no interferencia de dicho aserto I con cualquier sentencia atómica de S_i, es decir, se ha de poder probar el triple

$$NI(I, a) \equiv \{I \wedge pre(a)\}\ a\ \{I\}$$

para toda a sentencia atómica de S_i.

Ejemplo. Si tenemos un programa que sigue el paradigma del productor/consumidor y queremos probar su corrección parcial (esto es, que el programa hace lo que esperamos), suponiendo que ya tenemos demostrada las correcciones parciales del

productor y del consumidor, faltaría probar la no interferencia entre ambos. Para ello, y con el fin de evitar comprobar múltiples triples de Hoare, decidimos realizar la prueba mediante un invariante global, de foma que dicho invariante nos dé la corrección parcial de nuestro programa. Si conseguimos probar la invarianza de dicha propiedad, tendremos demostrada la corrección del programa.

Para la corrección parcial, queremos probar que el consumidor no lea una determinada variable x antes de que el productor no escriba en ella, y que el productor espere a que el consumidor lea el valor anterior de x antes de modificarlo. Debemos buscar una forma de representar esto matemáticamente, para así definir I como la relación matemática que cumpla esto, y proceder a demostrar que I es un invariante global, para así tener la corrección de nuestro programa probado.

Una forma de hacerlo es contando el número de veces que se lee y escribe en la variable:
Sea n el número de veces que el productor ha producido y escrito un valor en x y m el número de veces que el consumidor ha leído el valor de x, debe cumplirse que consumidor no lea x más veces que el productor haya escrito en ella, es decir, que

$$m \leqslant n$$

Así mismo, el productor no puede escribir en x más veces de las que el consumidor haya leido x, sino que sólo puede hacerlo una vez que este lo haya hecho. Es decir, si $m = n$, entonces el productor podrá volver a escribir en x, por lo que tendremos que $n = m+1$. Sin embargo, este no puede volver a escribir hasta que el consumidor haya leido la variable. De esta forma, limitamos al productor de generar más datos de las que el productor pueda consumir, con la expresión:

$$n \leqslant m + 1$$

En resumen, el buen funcionamiento del programa dependerá del invariante global:

$$I = m \leqslant n \leqslant m + 1$$

siendo m y n las variables anteriormente definidas.

Pasamos ahora a la demostración de la corrección de nuestro programa:

1. Primero, debemos comprobar que el invariante es cierto al inicio del programa.

   Inicialmente, el productor no ha escrito todavía en x, al igual que el consumidor no ha leido todavía x, por lo que:

   $$0 = m \leqslant n = 0 \leqslant m + 1 = 0 + 1$$

   Y vemos que se verifica I.

2. Posteriormente, debemos dividir el programa en bloques de forma que debemos probar que antes y después de cada bloque el invariante sigue siendo válido. Los bloques que consideraremos estarán formados por una iteración del productor y por una del consumidor.

Suponiendo que al inicio de un bloque se verifica I, es decir, que $m \leqslant n \leqslant m+1$, debemos probar que las la escritura de x por el productor y tras la lectura de x por el consumidor se verifica $m' \leqslant n' \leqslant m' + 1$, siendo m' y n' el número actualizado de veces que se ha leido y escrito en la variable, respectivamete.

Para ello, suponemos que se verifica I. Como demostramos anteriormente la corrección parcial de cacda programa (del productor y del consumidor), sabemos que tras un bloque (es decir, una iteración de cada uno de los dos programas), el productor habrá generado un dato que el consumidor habrá leido (o que el consumidor habrá leido un dato y el productor habrá generado el siguiente). En cualquier traza de ejecución, tendremos que:

$$m' = m + 1 \qquad n' = n + 1$$

Y ahora tendremos que comprobar que el invariante se sigue cumpliendo:

$$m \leqslant n \leqslant m + 1 \Longrightarrow m + 1 \leqslant n + 1 \leqslant m + 2 \Longrightarrow m' \leqslant n' \leqslant m' + 1$$

Notemos que hemos supuesto que se verificaba I al inicio del bloque. En principio nada nos garantiza esto, pero como anteriormente probamos que se cumplía al inicio del programa, se cumple antes de la primera ejecución del bloque, y como hemos demostrado que se cumple tras el bloque, se cumplirá antes de la segunda ejecución del bloque, ... Hemos realizado una especie de inducción en nuestro SLF.

Hemos demostrado ya que I es un invariante global. Como este nos garantiza la corrección de nuestro programa concurrente, tenemos ya demostrada la corrección del mismo.

1.5.6. Demostrar que un programa termina

Para demostrar que un programa concurrente que no realiza llamadas bloqueantes termina, es necesrio para ello el concepto de *vector variante*.

Definición 1.6 (Vector variante)**.** Dado un bucle en un lenguaje de programación:

```
1 while B do begin
     {Cuertpo del bucle}
  end
```

Donde la condición `B` tiene n variables que son modificadas dentro de dicho bucle: a_1, a_2, ..., a_n. Entonces, el vector variante para dicho bucle es un conjunto de n-uplas de la forma $(p_1, p_2, \ldots, p_n)$ de forma que p_i es un posible valor que puede tomar la variable a_i dentro del bucle, para $i \in \{1, \ldots, n\}$.

Si no admitimos la existencia de llamadas bloqueantes, la única forma en la que un programa puede no terminar es debido a la existencia de un bucle de forma que la condición de iteración B nunca sea falsa, por lo que dicho bucle permanecerá iterando por siempre, haciendo que nuestro programa no termine.

Por tanto, cuando nos pregunten demostrar que un programa $\{P\}\ S\ \{Q\}$ termina, deberemos demostrar que cada bucle que lo compone termina. Para ello:

1. Debemos primero identificar en cada bucle las variables de la condición de iteración B del bucle que adoptan distintos valores a lo largo del bucle.

2. Posteriormente, identificaremos cada uno de los posibles valores que pueden tomar cada una de esas variables, con lo que tendremos el vector variante del bucle. En caso de que la condición de salida ($\neg B$) no forme parte del vector variante, el bucle no terminará y por tanto el programa tampoco.

3. Si la condición de salida se encuentra en el vector variante, debemos razonar que en algún momento del bucle se alcanzará esta condición.

Ejemplo. Imponer condiciones iniciales sobre `n` para demostrar que el siguiente programa termina.

```
1 max = a[1]; i = 0;
  while (i <> n+1) do begin
     if (a[i] >= max) then max = a[i];
     i = i + 1;
5 end
```

1. La condición de iteración es $i \neq n+1$, y como la instrucción `i=i+1;` está presente en el cuerpo del bucle, la única variable de la condición de iteración que es modificada dentro del bucle es i.

2. El vector variante para dicho bucle contiene los posibles valores de i dentro del bucle, los cuales se corresponden con:

$$\{0, 1, 2, 3, \ldots\} = \mathbb{N}$$

3. Como i comienza el bucle en 0 y en cada iteración se incrementa una unidad, tendremos que $i = n+1$ en alguna iteración si $n+1 \in \mathbb{N} \iff n \in \mathbb{N} \cup \{-1\}$.

   Distinguiendo casos:

   - Si $n < -1$, entonces $n - 1 < 0$, por lo que $i_0 = 0 \neq n+1$ y como i se incrementa en una unidad, el programa no terminaría.
   - Si $n = -1$, entonces $i_0 = 0 = n+1$, por lo que el bucle no llegaría a ejecutarse y el programa terminaría.
   - Si $n > -1$, entonces $n - 1 > 0$ y como i se inicializa a 0 y se incrementa una unidad en cada iteración, la condición de salida sería cierta tras $n+1$ interaciones, que pueden hacerse en tiempo finito, luego el programa terminaría.

Finalmente, concluimos que el programa termina si, y solo si $n + 1 \in \mathbb{N}$.

2. 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.

Estudiaremos en profundidad el problema de la exclusión mutua, que ya obtuvo una solución en la asignatura de Arquitectura de Computadores, usando para ello cerrojos hardware o instrucciones máquina que nos proveían de funcionalidades deseadas a la hora de implementar una exclusión mutua.

Posteriormente, continuaremos con el problema de la sincronización de procesos en un programa concurrente, usando para ello conceptos con un mayor nivel de abstracción, los cuales nos permitirán resolver problemas más complejos de forma sencilla. Nos centraremos en el uso de los monitores, construcciones de alto nivel que nos ofrecen mayor libertad que los semáforos.

2.1. Problema de la exclusión mutua

En esta Sección tratamos de resolver el problema de la exclusión mutua mediante soluciones software, de forma que la solución no dependa del repertorio de instrucciones de una máquina, sino que sea una solución portable a cualquier dispositivo, de forma que podamos asegurar sobre los procesos de nuestros programas concurrentes todas las propiedades deseadas.

Consideraremos solo soluciones al problema en el que el acceso a la sección crítica se resuelva mediante instrucciones básicas de lectura y escritura sobre una o varias variables compartidas en memoria.

Como mecanismo para realizar la espera de los procesos en el acceso a la sección crítica usaremos la *espera ocupada*, es decir, meteremos a los procesos que no deben entrar a la sección crítica todavía en un bucle que realice iteraciones "vacías" (sin ninguna utilidad) con la finalidad a que esperen a que el proceso que se encuentre en la sección crítica abandone la misma y deje pasar al siguiente.

Hemos de comentar que la espera ocupada no es la mejor solución de espera para los procesos, ya que introduce un uso innecesario de los procesadores con el fin de

que ciertos procesos esperen. Puede considerarse una solución aceptable cuando el sistema no disponga de muchos procesos, pero en otro caso podríamos considerar otro tipo de esperas, como que el propio Sistema Operativo suspenda a los procesos.

2.1.1. Condiciones de Dijkstra

Dijkstra enunció que para obtener una solución parcialmente correcta al problema de la exclusión mutua, debían cumplirse 4 condiciones:

1. *No hacer ninguna suposición acerca de las instrucciones o número de procesos soportados por el multiprocesador.* Esto es, solo podremos hacer uso de operaciones que entendemos como básicas, tales como leer o escribir en una variable compartida para resolver el problema.

   Dichas instrucciones se ejecutarán de forma atómica, de forma que si dos procesos distintos intentan acceder a la vez a una misma posición de memoria, será el controlador de memoria quien determine de forma arbitraria qué proceso accederá antes y qué proceso después, de forma que el acceso a memoria se lleve a cabo secuencialmente, pero no de una forma predecible.

2. *No hacer ninguna suposición acerca de la velocidad de ejecución de los procesos*, salvo que esta no es cero, para que se cumpla la hipótesis de Progreso Finito.

3. *Cuando un proceso se encuentra ejecutando código fuera de la sección crítica, no puede impedir que otros entren a la misma.*

4. *La sección crítica será alcanzada finalmente por alguno de los procesos que quieran entrar.* Esta condición asegura la propiedad de *alcanzabilidad*, que excluye la posibilidad de que los procesos lleguen a una situación de interbloqueo.

   Esta propiedad no asegura que todos los procesos entren alguna vez a la sección crítica, y mucho menos que lo hagan de forma equitativa.

2.1.2. Método de refinamiento sucesivo

Dijkstra propuso a su vez una forma de obtener una solución al problema de la exclusión mutua para dos procesos, basada en 4 pasos, modificaciones o etapas a partir de un esquema inicial para obtener la solución de forma razonada, que terminará en una quinta etapa, denominada *algoritmo de Dekker*.

Primera etapa

Inicialmente, se presupone que los procesos alternarán su entrada en la sección crítica según indique el valor de una variable compartida llamada `turno`. Dicha variable contendrá el identificador del proceso que en cada momento puede entrar a la sección crítica.

En un escenario con dos procesos que se disponen a ejecutar una sección crítica, la primera etapa consta del código de la Figura 2.1.

```
var turno : integer;

Process P1();
begin
   while true do
   begin
      { Acceso a la SC }
      while turno <> 1 do
      begin
         null;
      end do
      { Sección crítica }
      turno := 2;
   end do
end
```

```


Process P2();
begin
   while true do
   begin
      { Acceso a la SC }
      while turno <> 2 do
      begin
         null;
      end do
      { Sección crítica }
      turno := 1;
   end do
end
```

Figura 2.1: Algoritmo para la primera etapa de refinamiento sucesivo.

Esta solución garantiza el acceso en exclusión mutua de los procesos a la sección crítica de forma independiente a la velocidad de ejecución de los mismos, por lo que la solución es segura. Sin embargo, no cumple la tercera condición de Dijkstra, ya que la solución obliga a la alternancia entre los procesos en la entrada a la sección crítica.

De esta forma, si el proceso $P1$ entra a la sección crítica mientras que $P2$ se encuentra haciendo otras operaciones independientes, el proceso $P1$ no podrá volver a entrar a la sección crítica hasta que no lo haga $P2$.

Segunda etapa

La alternancia que obtuvimos en la etapa anterior y que nos impedía cumplir con todas las condiciones de Dijkstra se debía a que para decidir qué proceso entraba en la sección crítica era necesario almacenar información global del estado del programa.

Para evitar esto, la idea ahora es asociar a cada proceso una variable que contenga su información de estado, variable que llamaremos `clave`, la cual indicará de forma binaria si el proceso se encuentra o no en la sección crítica en dicho instante de ejecución del algoritmo, mediante dos estados:

- Estado pasivo, el proceso no intenta acceder a la sección crítica, representado con un 1.
- El proceso intenta acceder a la sección crítica, representado con un 0.

El código quedaría como el de la Figura 2.2.

Como podemos ver, el protocolo de entrada consiste en leer el valor de la clave del otro proceso con la finalidad de consultar si dicho proceso ha entrado o no en la sección crítica, y esperar mientras este se encuentre dentro de la sección crítica.

Sin embargo, en este caso la solución no es segura, ya que si $P1$ y $P2$ se ejecutan a la misma velocidad, entonces ambos entrarían a la vez a la sección crítica, debido

```
var c1, c2 : integer;

Process P1();
begin
   c1 := 1;

   while true do
   begin
      { Acceso a la SC }
      { Si P2 entró }
      while c2 = 0 do
      begin
         null;
      end do

      { Entra a la sección crítica }
      c1 := 0;
      { Sección crítica }
      c1 := 1;
   end do
end
```

```


Process P2();
begin
   c2 := 1;

   while true do
   begin
      { Acceso a la SC }
      { Si P1 entró }
      while c1 = 0 do
      begin
         null;
      end do

      { Entra a la sección crítica }
      c2 := 0;
      { Sección crítica }
      c2 := 1;
   end do
end
```

Figura 2.2: Algoritmo para la segunda etapa de refinamiento sucesivo.

a que los dos verían que el estado del otro es pasivo, con lo que ninguno entraría en el bucle de espera ocupada. Notemos que esto sucede porque cambiamos el estado de un proceso a 0 justo antes de entrar a la sección crítica, por lo que es ya tarde para impedir la entrada a otro proceso.

Como la bondad de la solución depende de la velocidad de ejecución relativa entre los procesos, se dice que es inaceptable por incumplir la segunda condición de Dijkstra.

Tercera etapa

Para esta etapa planteamos una sencilla modificación sobre la etapa anterior, que consiste en cambiar el valor de la variable clave a 0 antes de consultar el valor de la variable clave del otro proceso. De esta forma, para que un proceso pueda entrar a la sección crítica, debe primero cambiar su estado a 0, con el fin de recuperar la condición de seguridad de que solo un proceso pueda entrar a la vez a la sección crítica. El código resultante es el de la Figura 2.3.

Sin embargo, si ambos procesos tienen la misma velocidad, puede suceder que ambos cambien el valor de su clave a 0 al mismo tiempo, con lo que se de una situación de interbloqueo, que incumpliría la cuarta condición de Dijkstra, al no poder alcanzar nunca la sección crítica.

Cuarta etapa

Lo que causó el problema en la tercera etapa fue que puede suceder que un proceso cambie el valor de su clave a la vez que el otro de forma concurrente, sin que

```
var c1, c2 : integer;

Process P1();
begin
   c1 := 1;

   while true do
   begin
      { Acceso a la SC }
      c1 := 0;
      { Si P2 entró }
      while c2 = 0 do
      begin
         null;
      end do
      { Sección crítica }
      c1 := 1;
   end do
end
```

```

Process P2();
begin
   c2 := 1;

   while true do
   begin
      { Acceso a la SC }
      c2 := 0;
      { Si P1 entró }
      while c1 = 0 do
      begin
         null;
      end do
      { Sección crítica }
      c2 := 1;
   end do
end
```

Figura 2.3: Algoritmo para la tercera etapa de refinamiento sucesivo.

este se de cuenta de que el otro lo ha hecho a la vez. La solución que se propone en esta etapa es permitir a un proceso volver a cambiar el valor de su clave a 1 si después de asignar su clave a 0, comprueba que el otro proceso también cambió su clave al mismo valor. De esta forma, planteamos el código de la Figura 2.4.
Sin embargo, si ambos procesos se ejecutasen a la misma velocidad, se podría seguir produciendo un interbloqueo entre ambos procesos, aunque esta situación ahora sea más improbable. La solución no sería válida por incumplir tanto la segunda como la cuarta condición de Dijkstra.

La conclusión a la que llegamos tras todas estas etapas es que las variables `c1` y `c2` nos son útiles para coordinar la entrada a la sección crítica, pero no son suficientes para dar una solución correcta al problema que tratamos de resolver.

2.1.3. Algoritmo de Dekker

Se podría considerar como una quinta etapa del método de refinamiento sucesivo, pero esta vez obteniendo una solución válida del problema. El algoritmo de Dekker junta las ideas presentes en la primera y cuarta etapa de refinamiento de Dijkstra:

- La primera etapa producía una solución segura, pero obligaba a la alternancia en el acceso de los procesos a la sección crítica.

- Por otra parte, la cuarta etapa no cuenta con dicha alternancia en el acceso, pero puede llevar a un interbloqueo de los procesos del programa concurrente.

```
var c1, c2 : integer;

Process P1();
begin
   c1 := 1;

   while true do
   begin
      { Acceso a la SC }
      c1 := 0;
      { Si P2 entró }
      while c2 = 0 do
      begin
         c1 := 1;
         while c2 = 0 do
         begin
            null;
         end do
         c1 := 0;
      end do
      { Sección crítica }
      c1 := 1;
   end do
end
```

```


Process P2();
begin
   c2 := 1;

   while true do
   begin
      { Acceso a la SC }
      c2 := 0;
      { Si P1 entró }
      while c1 = 0 do
      begin
         c2 := 1;
         while c1 = 0 do
         begin
            null;
         end do
         c2 := 0;
      end do
      { Sección crítica }
      c2 := 1;
   end do
end
```

Figura 2.4: Algoritmo para la cuarta etapa de refinamiento sucesivo.

Para resolver el problema, se considera el código de la cuarta etapa de Dijkstra y se le añade un orden establecido en la entrada mediante una variable `turno`, para desempatar la situación en la que los dos procesos quieran entrar exactamente al mismo tiempo en la sección crítica.

De esta forma, un proceso que quiera entrar en la sección crítica asignará primero su clave a 0, y si el otro proceso también tiene su clave a 0, lo primero que hará es comprobar de quién es el turno y si no dispone del mismo, cambiará su clave a 1 pasando a esperar y dejando al otro proceso continuar con la ejecución de la sección crítica. Vemos el código en la Figura 2.5.

```
var c1, c2, turno : integer;

Process P1();
begin
   while true do
   begin
      { Acceso a la SC }
      c1 := 0;

      while c2 = 0 do
      begin
         if turno = 2 then
         begin

            c1 := 1;
            while turno = 2 do
            begin
               null;
            end do
            c1 := 0;

         end
      end do

      { Sección crítica }
      turno := 2;
      c1 := 1;
   end do
end
```

```


Process P2();
begin
   while true do
   begin
      { Acceso a la SC }
      c2 := 0;

      while c1 = 0 do
      begin
         if turno = 1 then
         begin

            c2 := 1;
            while turno = 1 do
            begin
               null;
            end do
            c2 := 0;

         end
      end do

      { Sección crítica }
      turno := 1;
      c2 := 1;
   end do
end
```

Figura 2.5: Algorimo de Dekker.

Propiedades de corrección

En esta sección, demostraremos que se cumple siempre el acceso en exclusión mutua a la sección crítica por parte de los procesos que intervienen en los programas concurrentes, así como la propiedad de alcanzabilidad de la sección crítica y vivacidad de los procesos que tratan de hacer uso de la misma:

Exclusión mutua.

El proceso P_i (con $i = 1$ ó $i = 2$) entrará en la sección crítica solo si el otro proceso, P_j mantiene su clave cj a 1. Dado que la clave de un proceso solo la puede modificar el propio proceso y que el proceso P_i comprueba la clave cj solo después de asignar su propia clave ci a 0, si el proceso P_i entra en sección crítica, se ha de cumplir la condición $ci = 0 \wedge cj = 1$. Notemos que esta situación es incompatible con la condición de que el proceso P_j entre en la sección crítica: $cj = 0 \wedge ci = 1$.

Alcanzabilidad de la sección crítica.

Para demostrar la alcanzabilidad de la sección crítica, distinguimos casos:

- Si suponemos que el proceso P_i intenta entrar solo en la sección crítica, entonces el otro proceso P_j se mantendrá en estado pasivo, con lo que el valor de su clave cj será 1. De esta forma, el proceso P_i puede entrar a la sección crítica.
- Sin embargo, si tanto P_i como P_j intentan entrar a la vez a la sección crítica y suponemos que $turno = i$, entonces:
  - Si P_j encuentra la clave ci a 1, entonces P_j entrará en la sección crítica.
  - Si P_j encuentra la clave ci a 0, como $turno = i$, entonces P_j entrará en el segundo bucle interno para realizar la espera ocupada, poniendo antes su clave cj a 1, que permitirá pasar al proceso P_i.
  - Si P_i encuentra la clave cj a 1, entonces P_i entrará en la sección crítica.
  - Si P_i encuetran la clave cj a 0, se mantendrá realizando iteraciones en el bucle de espera ocupada más externo con ci a 0, hasta que lea el valor de cj a 1, que sucederá por el punto superior, con lo que P_i entrará en la sección crítica.

Vivacidad.

Dependiendo del hardware de control de acceso a memoria, el algoritmo de Dekker puede llegar a provocar la inanición de uno de los dos procesos:

Supongamos que tenemos a los procesos $P1$ y $P2$ ejecutando su código, queriendo acceder continuamente a la sección crítica. Supongamos además que el proceso $P2$ se ejecuta a una velocidad bastante lenta en comparación al proceso $P1$. Nos encontramos en el caso en el que ambos procesos cambiaron sus claves al mismo tiempo y el turno inicial era 1, con lo que $P1$ pasó a ejecutar el código de la sección crítica y $P2$ se quedó esperando en el bucle más interno, con el valor de su clave `c2` a 1.

Debemos recordar que anteriormente mencionamos que el acceso al módulo de memoria no se hace de forma paralela, sino que se hace de forma secuencial, de forma que si dos procesos intentan acceder a la vez a una misma posición de memoria es el controlador de memoria quien determina el acceso a un proceso de forma arbitraria.

Supongamos pues que $P1$ termina de ejecutar el código de la sección crítica, con lo que cambia el valor de la variable `turno` a 2, `c1` a 1 y cambia también `c1` a 0. Posteriormente, como $P1$ cambió `turno` a 2, el proceso $P2$ sale del bucle más interno, con lo que se dispone a cambiar el valor de su clave a 0.

Sin embargo, en este momento sucede que tanto $P1$ como $P2$ intentan acceder a la vez al valor de `c2`, $P1$ para leer (en la condición del `while` exterior) y $P2$ para escribir. Si en dicho momento el controlador de memoria da prioridad a las lecturas, $P1$ volvería a introducirse en la sección crítica.

Inmediatamente, $P2$ se dispondría a cambiar el valor de su clave a 0, pero como mencionamos anteriormente, $P2$ es muy lento, con lo que resulta que le da tiempo a $P1$ a ejecutar la sección crítica y volver a la lectura de `c2` en el bucle más externo a la vez que $P2$, con lo que el controlador de memoria puede volver a darle prioridad.

Si este escenario sucede de forma indefinida, tenemos una falta de vivacidad en el proceso $P2$, ya que mientras $P1$ esté en funcionamiento, no podrá avanzar en su ejecución.

Equidad del protocolo.
Como hemos comentado en el apartado de vivacidad, la equidad del algoritmo de Dekker dependerá de la equidad del hardware de la máquina en el que ejecutemos el programa concurrente. Si existen peticiones de acceso simultáneo a una misma dirección de memoria compartida por dos procesos, uno para lectura y otro para escritura de forma que el hardware da prioridad a las lecturas, no se puede demostrar que el algoritmo de Dekker sea equitativo, pudiendo llegar al escenario de inanición de uno de los procesos como ya se ha descrito anteriormente.

2.1.4. Algoritmo de Dijkstra

Una vez visto el algoritmo de Dekker, primera solución aceptable al problema de la exclusión mutua (aunque no cumpla con las propiedades deseables de vivacidad y equidad), nos encontramos con que no es generalizable para n procesos, con lo que mostramos a continuación el algoritmo de Dijkstra, que resuelve el problema de la exclusión mutua para n procesos utilizando un array de n posiciones en las que cada proceso almacena su estado, destacando tres posibles estados:

- Proceso pasivo, el proceso no intenta acceder al protocolo de entrada.
- Solicitando, el proceso intenta acceder al protocolo de entrada.
- En SC, el proceso está dentro de la sección crítica.

El algoritmo de Dijkstra podemos verlo en la Figura 2.6.

```
var c : array[0..n-1] of (pasivo, solicitando, en_SC);
    turno : 0..n-1;

Process Pi();
begin
   while true do
   begin
      { Acceso a la sección crítica }
      repeat
         c[i] := solicitando;

         { A }
         while turno <> i do
         begin
            if c[turno] = pasivo then turno := i;
         end do

         c[i] := en_SC;

         { B }
         j := 0;
         while (j<n) and (j=i or c[j] <> en_SC) do
         begin
            j := j+1;
         end do
      until j >= n

      { Sección crítica }
      c[i] := pasivo;
   end do
end
```

Figura 2.6: Algoritmo de Dijkstra, código para el i-ésimo proceso.

De esta forma, lo primero que hace un proceso P_i al llegar al protocolo de entrada a la sección crítica es cambiar su estado a `solicitando`. Posteriormente:

- Si es su turno, no realizará ninguna iteración del bucle `A`, con lo que pasará inmediatamente a la ejecución del bucle `B`, tras cambiar su clave a `en_SC`.
- Si no es su turno:
  - Si el proceso que tiene el turno está en estado pasivo, entonces el proceso P_i pone el turno a i, siendo ahora su turno, con lo que pasa del bucle `A`.
  - Si el proceso que tiene el turno no está en estado pasivo, entonces el proceso P_i esperará hasta que este lo esté, es decir, hasta que el proceso P_{turno} abandone la sección crítica.

Una vez superado el bucle `A`, entonces el estado del proceso P_i pasará a `en_SC`, y solo tendrá que superar el bucle `B` para poder entrar a la sección crítica.

Puede sueceder que dos (o más, en cuyo caso es una explicación análoga) procesos, P_i y P_j lleguen al bucle `A` de forma que (suponiendo que `c[turno] = pasivo`)

ambos lo ejecuten a la misma velocidad, con lo que ambos pasen dicho bucle quedando la variable `turno` asignada al identificador de cualquiera de los dos procesos (a i o a j). Posteriormente, ambos cambiarán su estado a `en_SC`. En dicho caso, contamos con el bucle `B` para cumplir con la condición de seguridad de tener un único proceso ejecutando la sección crítica a la vez.

Lo que hace el bucle `B` es comprobar todos los estados de los procesos antes de dejar pasar al proceso P_i entrar a la sección crítica, de forma que si todos los demás procesos tienen un estado distinto de `en_SC`, entonces el contador `j` llegará hasta `n`, con lo que P_i saldrá del bucle `B` así como del `repeat` de entrada a la sección crítica.

En el caso en el que dos (o más) procesos estén con estado `en_SC`, entonces la variable `j` de cada proceso no llegará a aumentar hasta `n` (lo que le permitiría salir del bucle `repeat` y entrar en sección crítica), con lo que los procesos no podrán salir del `repeat`, teniendo que volver a pasar por el bucle `A`, donde el turno estará ya fijado en un proceso con un estado distinto de pasivo (el último que actualizó el valor de dicha variable).

Propiedades de corrección

Pasamos ahora a demostrar las propiedades de corrección del algoritmo de Dijsktra:

Exclusión mutua.

La propiedad de exclusión mutua está garantizada gracias al bucle `B`:
El proceso P_i (con $i \in \{0, \ldots, n-1\}$) entrará en la sección crítica solo si el resto de procesos P_j con $j \neq i$ tienen su estado distinto de `en_SC`. En dicho caso, P_i (y solo él) pasará a ejecutar la sección crítica, poniendo su estado a `en_SC`.

Supuesto que ahora otro proceso P_j intenta acceder a la sección crítica, solo podrá hacerlo si el resto de procesos tienen su estado distinto de `en_SC`, algo que no puede suceder hasta que el proceso P_i salga de la sección crítica.

Alcanzabilidad de la sección crítica.

Supuesto que disponemos de m procesos $P_1, P_2, \ldots, P_m$ de forma que tratan de acceder a la sección crítica al mismo tiempo, entonces el valor de la variable compartida `turno` será un cierto $k \in \{1, \ldots, m\}$, correspondiente al identificador del último proceso que cambió su valor.

Los procesos que superen el bucle `A` junto con P_k no podrán superar el bucle `B` con un valor de `j` igual o superior a `n`, por lo que todos los procesos (junto con P_k) volverán al bucle `A`.

En dicho instante, el turno estará fijado a k y `c[k]` tendrá un valor distinto de pasivo, con lo que el proceso P_k será ahora el único que consiga pasar el bucle `A`, con lo que completará el bucle `B` con un valor de `j` igual a `n`, lo que le permitirá acceder a la sección crítica.

Vivacidad.

El algoritmo de Dijkstra también puede llevar a la inanición de uno de los procesos del programa concurrente, al igual que el algoritmo de Dekker.

Para ver dicha situación, supongamos que tenemos tres procesos, P_0, P_1 y P_2, los tres intentando acceder a la sección crítica. Supongamos que inicialmente tenemos $turno = 0$ y que la traza de ejecución obtenida es aquella que hace que P_0 salte el bucle `A`, antes de que P_1 y P_2 cambien su estado a `solicitando`, con lo que ambos quedarán bloqueados en dicho bucle.

Esto permite a P_0 entrar en la sección crítica. Cuando este proceso salga y cambie su estado a `pasivo`, provocará que los dos procesos anteriores pasen el bucle `A`, en un orden que puede ser primero P_2 (cambia $turno = 2$) y luego P_1 (cambia $turno = 1$), de forma que ambos lleguen al bucle `B`, que obligue a ambos a pasar nuevamente por el bucle `A`, donde el proceso P_2 quedará bloqueado y será P_1 quien consiga llegar a la sección crítica.

Suponiendo que ahora P_0 vuelve a intentar acceder a la sección crítica y que después P_1 cambia su estado a `pasivo` (sale de la sección crítica), puede suceder la situación anterior pero en este caso con P_0, es decir, que sea P_2 quien salga antes del bucle `A` y P_0 después, con lo que el turno queda asignado finalmente a 0, lo que permitirá a P_0 volver a entrar en la sección crítica.

Esta situación se puede repetir tantas veces como queramos, de forma que P_2 nunca llegue a ejecutar la sección crítica, por lo que no es posible demostrar la vivacidad del algoritmo de Dijkstra, ya que este puede llevar a la inanición de uno de los procesos que ejecutan el protocolo de entrada a la sección crítica.

Equidad.
Como no podemos garantizar la vivacidad de la solución, mucho menos podremos garantizar la equidad de los procesos en el acceso a la sección crítica.

2.1.5. Algoritmo de Knuth

Con la finalidad de no caer en los errores de los algoritmos de Dekker y Dijkstra (los cuales pueden llevar a un estado en el que un proceso sufra inanición), Donald Knuth añadió una quinta condición a las condiciones de Dijkstra para la corrección parcial de una solución al problema de la exclusión mutua:

5. *Se tiene que poder demostrar que todos los procesos pueden sufrir un retraso máximo en el acceso a la sección crítica que sea calculable.*

   Con el fin de garantizar que ningún proceso sufrirá de inanición, ya que en algún momento conseguirá entrar a la sección crítica, lo que nos dé la propiedad de vivacidad de su solución.

Donald Knuth se dió cuenta de que la propiedad de vivacidad no podía ser demostrada debido a que la variable compartida `turno` utilizada en los algoritmos de Dekker y Dijkstra sufría condiciones de carrera. El algoritmo propuesto por Knuth hace uso de una variable local `j` que copia el valor de la variable compartida `turno`, con el fin de evitar dichas condiciones de carrera. Por lo demás, el algoritmo es similar al de Dijkstra, tal y como podemos ver en la Figura 2.7.

```
var c : array[0..n-1] of (pasivo, solicitando, en_SC);
    turno : 0..n-1;

Process Pi();
var j : 0..n-1;
begin
   while true do
   begin
      repeat
         c[i] := solicitando;
         j := turno;

         { A }
         while j <> i do
         begin
            if c[j] <> pasivo then j := turno;
            else j := (j-1) mod n;
         end do

         c[i] := en_SC;
         k := 0;

         { B }
         while (k<n) and (k=i or c[k] <> en_SC) do
         begin
            k := k+1;
         end do
      until k >= n;

      turno := i;
      { Sección crítica }
      turno := (i-1) mod n;
      c[i] := pasivo;
   end do
end
```

Figura 2.7: Algoritmo de Knuth.

De esta forma, suponiendo que somos el proceso P_i y que nos disponemos a ejecutar la sección crítica:
Si tenemos que `turno = i`, entonces P_i se salta el bucle `A`. Si no (`turno != i`), entonces:

- Si el proceso que tiene el turno no está en estado pasivo, entonces el proceso se bloqueará realizando una espera activa, copiando el valor de `turno` en `j`. De esta forma, cuando el proceso que tenía el turno salga de la sección crítica, cambiará el valor de `turno` y el proceso cambiará su copia en `j`, con lo que saldrá de la espera ocupada e intentará volver a realizar el protocolo de entrada para la sección crítica.

- Si el proceso que tiene el turno está en estado pasivo, entonces la copia local del valor de `turno`, es decir, la variable `j` ciclará su valor, de forma que volvamos a ejecutar el protocolo de entrada, destacando tres posibilidades:

- Si al ciclar el valor de `j` ahora tenemos que `j = i`, entonces P_i pasa el bucle `A`.
- Si al ciclar el valor de `j` se detecta que P_j no está en estado pasivo, entonces P_i realizará una espera activa, de forma que actualice `j` a `turno`, luego ciclará el valor de `j` y detectará que P_j no estará en estado pasivo.
- Si al ciclar el valor de `j` se detecta que P_j está en estado pasivo, se cicla el valor de `j` otra vez, volviendo a destacar tres posibilidades.

De esta forma, Knuth controla el número de veces que los procesos pueden adelantarse los unos a los otros dentro del bucle `A`.

Por el resto, es idéntico al funcionamiento del algoritmo de Dijkstra, salvo que solo modifica el valor de la variable `turno` una vez el proceso se encuentra dentro de la sección crítica, evitando condiciones de carrera.

Propiedades de corrección

Veamos las propiedades de corrección del algoritmo de Knuth:

Exclusión mutua.
Está garantizada gracias al bucle `B`, tal y como razonamos en el algoritmo de Dijkstra.

Alcanzabilidad de la sección crítica.
Supuesto que tenemos m procesos, P_1, P_2, ..., P_m que tratan de acceder a la sección crítica a la vez, y suponiendo que `turno = k`:

- Si $k \in \{1, \ldots, m\}$, entonces, P_k superará el bucle `A` y tratará de entrar a la sección crítica, distinguiendo dos posibilidades:
  - Si el resto de procesos vieron que `c[k]` era distinto de pasivo, los $m - 1$ procesos restantes quedaron bloqueados en el bucle `A`, con lo que solo el proceso P_k tendrá su clave a `en_SC`, con lo que conseguirá superar el bucle `B`, entrando a la sección crítica.
  - Si hay algún proceso (o varios) que vieron `c[k]` a pasivo (esto es, antes de que el proceso k cambiase el valor de su clave), estos irán ciclando el valor de `j`, de forma que puede haber algún conjunto de procesos que supere el bucle `A` junto con P_k. En dicho caso, tras superar el bucle `B`, ninguno de esos obtendrá `k = n`, con lo que han de volver a iterar en el bucle de `repeat`, realizando otra vez el bucle `A`. En este caso, estamos en las condiciones del punto superior, con lo que P_k accede a la sección crítica.
- Si $k \notin \{1, \ldots, m\}$, entonces tenemos que `c[k] = pasivo`. En dicho caso, todos los procesos irán ciclando su valor de `j` decrementándolo, de forma que si algún proceso P_i consigue tener $i = j$, entonces superará el bucle `A` y destacamos dos posibilidades:
  - Si solo un proceso P_i consigue tener $i = j$, entonces conseguirá superar el bucle `B` con $k = n$, con lo que entrará a ejecutar la sección crítica.

- Si hay dos o más procesos que consiguieron tener $i = j$ (puede suceder, si los procesos con i más lejana a `turno` ejecutaron el bucle `A` de forma más rápida), al superar el bucle `B`, ninguno conseguirá tener $k = n$, con lo que todos volverán a ejecutar el bucle `A`. En este momento, todos estos procesos que tenían otro proceso más cercano a `turno` quedarán bloqueados en el bucle `A`, ya que observarán que `c[j] != pasivo` en el proceso con el índice j más cercano a `turno`, por lo que solo el proceso P_i con índice más cercano a `turno` conseguirá superar el bucle `A`, quien superará el bucle `B` con un valor $k = n$, lo que le permitirá ejecutar la sección crítica.

Vivacidad.

Para demostrar la vivacidad del algoritmo de Knuth lo que tenemos que hacer es demostra la ausencia de inanición, es decir, que no podemos tener un proceso que nunca alcance la sección crítica.

Por reducción al absurdo, supongamos pues, que en un escenario con n procesos tenemos un proceso $P_{victima}$ (con $victima \in \{1, \ldots, n\}$) que nunca puede alcanzar la sección crítica, debido a que existe un proceso P_i (con $i \neq victima$) que adelanta a $P_{victima}$ de forma indefinida en el acceso a la sección crítica.

Entonces, ha de existir un proceso P_k con el índice k posterior a i pero anterior a $victima$ en el protocolo de entrada al monitor[1], ya que si no (esto es, $P_{victima}$ es el siguiente en turno rotatorio tras P_i) tras salir P_i de la seccion crítica ejecutaría la sentencia `turno := (i-1) mod n`, lo que provocaría que el siguiente proceso a ejecutar la sección crítica fuera $P_{victima}$, pero habíamos supuesto que $P_{victima}$ era adelantado de forma indefinida por el proceso P_i.

De esta forma, tenemos que también P_k adelanta de forma indefinida al proceso $P_{victima}$ (porque P_i lo adelanta de forma indefinida y el turno tras P_i pasa a un proceso posterior, de forma que P_k se posiciona antes de que el turno llegue a $P_{victima}$), con lo que se encuentra en la misma situación del proceso P_i anterior, pudiendo encontrar un proceso P_h con índice posterior a j pero anterior a $victima$ que también adelante a $P_{victima}$ de forma indefinida.

Como podemos observar, podemos repetir este razonamiento tantas veces como queramos, pudiendo obtener una cantidad de procesos superior a n que ejecuta el algoritmo de Knuth. Sin embargo, como la cantidad de procesos que ejecuta el algoritmo no puede ser superior a n, llegamos a contradicción con que $P_{victima}$ era un proceso que nunca puede entrar a la sección crítica, lo que demuestra la propieda de vivacidad del algoritmo de Knuth.

Equidad.

Para demostrar la propiedad de equidad del algoritmo de Knuth lo que haremos será acotar el número máximo de adelantamientos por otros procesos que puede sufrir un proceso que quiera entrar a la sección crítica. Para ello, sea $f : \mathbb{N} \to \mathbb{N}$ una función que contabiliza el número máximo de adelantamientos, $f(n)$ para un escenario con n procesos, entonces:

[1] Notemos que en este punto no estamos suponiendo de forma implícita que $i > k > victima$, ya que hablamos en sentido rotatorio, con lo que podemos tener $victima > i$ pero en dicho caso tendremos $k > victima$. Simplemente indicamos que el turno pasa antes a k por i que a $victima$.

- $f(0) = 0$, ya que en un escenario con ningún proceso, ningún proceso puede ser adelantado.
- $f(1) = 0$, ya que en un escenario con un proceso, este no puede ser adelantado por ningún otro.
- $f(2) = 1$, ya que si tenemos dos procesos, puede suceder que nuestro proceso sea adelantado por el otro proceso.

Para calcular $f(3)$ mostraremos el siguiente razonamiento, que nos permitirá generalizarlo, para conseguir obtener $f(n)$ a partir de $f(n-1)$, obteniendo una recurrencia y pudiendo calcular el valor de $f(n)$ para cualquier $n \in \mathbb{N}$.

Para ello, hemos de buscar cuál es la peor situación que se pueda dar en un escenario con 3 procesos de forma que un proceso obtenga el mayor número de adelantamientos en su entrada a la sección crítica. No demostraremos que se trata de la peor situación, pero puede intuirse que no hay peor situación que la que estamos a punto de describir:

Supongamos que tenemos 3 procesos: P_0, P_1, P_2 de forma que inicialmente $turno = 1$ y P_2 será nuestro proceso víctima (el que vamos a intentar que sea adelantado el máximo número de veces por P_0 y por P_1). Para ello, supongamos que los tres intentan acceder de forma simultánea al protocolo de entrada de la sección crítica, obteniendo la siguiente traza de ejecución:

- El proceso P_0 es el primero que cambia su clave `c[0]` a `solicitando`, de forma que cuando llega al bucle `A`, observa que `c[1]` está pasivo, decrementa el valor de `j` (ahora, $j = 0$), con lo que el proceso P_0 pasa del bucle `A`.
- Ahora, el proceso P_1 cambia su clave a `solicitando` y al ser $turno = 1$, pasa del bucle `A`.
- En este momento, P_2 cambia su clave a `soclitando` y como `c[1] = solicitando`, el proceso se queda bloqueado en el bucle `A`.
- Supongamos que P_0 es muy rápido y le da tiempo a asignarse el estado `en_SC`, de forma que antes de que P_1 cambie su clave a `en_SC`, le da tiempo a comprobar que no hay ningún otro proceso con clave `en_SC`, con lo que entra en la sección crítica. Ejecuta dicho código muy rápido, cambiando *turno* a 2 y regresando al bucle `A` para volver a entrar en la sección crítica.
- Si en este momento el proceso P_1 cambia su clave a `en_SC` y le da tiempo a completar el bucle `B` antes de que el proceso P_2 llegue a salir del bucle `A`, el proceso P_1 conseguirá salir del bucle `B` con $k = n$, lo que le permita ejecutar la sección crítica.
- Cuando el proceso P_1 abandone la seccion crítica, dejará *turno* a 0, con lo que P_2 seguirá bloqueado en el bucle `A` y será ahora P_0 quien vuelva a ejecutar la sección crítica.
- Finalmente P_0 sale de la sección crítica y cambia *turno* a 2, lo que ahora sí permitirá a P_2 entrar a la sección crítica.

De esta forma, el proceso P_2 ha sido adelantado 3 veces, obteniendo finalmente que los identificadores de los procesos que han ejecutado la sección crítica han sido:

0 1 0 2

Concluimos que $f(3) = 3$.

Si ahora disponemos que n procesos: P_0, P_1, ..., P_{n-1} de forma que inicialmente $turno = n - 2$ y P_{n-1} será nuestro proceso víctima, podemos obtener la siguiente traza de ejecución, de forma análoga al caso para 3 procesos:

- Inicialmente, todas las claves están en `pasivo`, ya que los n procesos accederán a la vez al protocolo de entrada de la sección crítica.
- El proceso P_{n-3} verá a todos pasivos, con lo que decrementará su índice en sentido rotatorio hasta llegar a tener $j = n - 2$, con lo que pasará del bucle `A`.
- P_{n-4} tendrá un comportamiento similar a P_{n-3}, y así con todos los procesos hasta P_0, quedando P_{n-1} y P_{n-2} que todavía están en estado pasivo.
- Si ahora P_{n-2} accede antes al bucle `A`, como tiene el turno, saltará el bucle sin problema y será el proceso P_{n-1} quien quedará bloqueado en el mismo.
- Buscamos el peor caso, luego el momento en el que $turno = n-1$ y P_{n-1} pueda ejecutar la sección crítica queremos alejarlo tanto como podamos. Para ello, buscamos el peor momento para tener $turno = n-2$, con lo que el turno ha de ciclar por todos los procesos anteriores para luego llegar a $n - 1$.

  De esta forma, suponemos que el acceso a la sección crítica ha sido aquél que nos daba el mayor número de adelantamientos para el caso $n - 1$ (si pensamos que $n = 4$, el acceso que nos daba el mayor número de adelantamientos para $n - 1 = 3$ era P_0 P_1 P_0). Después de este acceso, el proceso P_{n-2} podrá entrar a la sección crítica, con lo que el proceso P_{n-1} lleva ya $f(n - 1) + 1$ adelantamientos (los adelantamientos realizados al proceso P_{n-2} y el propio adelantamiento de P_{n-2}).
- Una vez salga el proceso P_{n-2} de la sección crítica, asignará *turno* a $n - 3$, y queremos buscar el peor caso de ejecución para el proceso P_{n-1}. Para ello, observemos que si tenemos $turno = n - 3$, nos encontramos en las hipótesis de poder encontrar nuevamente el peor caso de ejecución para que entre de nuevo P_{n-2}, pero ahora este no conseguirá entrar, ya que después del último proceso que adelante a P_{n-2} (que será P_0), este cambiará el turno a $n - 1$ (en vez del caso $n - 2$), con lo que después de tener otra vez los adelantamientos correspondientes al caso $n - 1$, el proceso P_{n-1} conseguirá entrar a la sección crítica.

  De esta forma, a la cantidad anterior de $f(n-1)+1$ adelantamientos, hemos de sumar nuevamente los adelantamientos que se realizan al proceso P_{n-2} en el caso para $n - 1$ procesos, es decir, $f(n - 1)$. En este punto, P_{n-1} será capaz de entrar en la sección crítica.

Por tanto, concluimos que el número máximo de adelantamientos en el caso n es:

$$f(n) = f(n-1) + 1 + f(n-1) = 2f(n-1) + 1$$

Recurrencia que podemos resolver[2], obteniendo que:

$$f(n) = 2^{n-1} - 1 \qquad \forall n \geqslant 1$$

Con lo que el número máximo de adelantamientos que puede sufrir un proceso en el acceso a la sección crítica está acotado, lo que demuestra la equidad del algoritmo.

Ejemplo. Observemos que lo que estamos haciendo en la última demostración de la propiedad de equidad es encontrar las peores trazas de ejecución de forma que un proceso sufra el máximo retardo posible en la entrada a la sección crítica. Hemos podido encontrar un patrón repetitivo en estas trazas, el cual está de forma implícita en la demostración y que ahora mostramos de forma explícita (suponiendo que en el caso n, es el proceso P_{n-1} quien sufre el número de adelantamientos $f(n)$):

Para $n = 0$: No hay adelantamientos posibles.

Para $n = 1$: No hay adelantamientos posibles.

Para $n = 2$: P_0.

Para $n = 3$: $P_0\ P_1\ P_0$.

Para $n = 4$: $P_0\ P_1\ P_0\ P_2\ P_0\ P_1\ P_0$.

Para $n = 5$: $P_0\ P_1\ P_0\ P_2\ P_0\ P_1\ P_0\ P_3\ P_0\ P_1\ P_0\ P_2\ P_0\ P_1\ P_0$.

Para n: (los que adelantan a P_{n-2}) P_{n-2} (los que adelantan a P_{n-2})

2.1.6. Algoritmo de Peterson para 2 procesos

Peterson propuso una forma simple de resolver el problema de la exclusión mutua para dos procesos, de forma que a partir de entonces se consideró la solución canónica a dicho problema. Esta es generalizable a n procesos.

La idea es la que se aplica en una pescadería: los clientes que quieren comprar pescado (en nuestro caso, los dos procesos que quieren entrar a la sección crítica), van asignándose sus turnos en orden de llegada, de forma que el último que se asigna el turno es quien debe esperar a que el otro pase primero (a la sección crítica). Una vez que este primero haya terminado, avisa al siguiente de que ya puede avanzar.

De esta forma, el algoritmo cuenta con las variables compartidas:

```
1 var c : array[1..2] of boolean;
      turno : 1..2;
```

[2]Tal y como aprendimos en asignaturas anteriores

Donde el array de claves `c` se inicializa con todos sus valores a `false`, de forma que `c[i]` indica si el i-ésimo proceso está o no solicitando acceder a la sección crítica. La variable `turno` sirve para resolver conflictos en el caso de que ambos procesos intenten acceder a la vez a la sección crítica.

Cuando un proceso quiere acceder a la sección crítica, lo que hace primero es cambiar su clave a "solicitando sección crítica" (representado con `true`). Posteriormente, se asigna el turno a dicho proceso. En el caso en el que ambos intenten acceder a la vez a la sección crítica, uno de ellos habrá sido necesariamente el último en asignarse el turno, y será el proceso que espere a que el otro ejecute la sección crítica.

```
Process P1();
begin
   while true do
   begin
      { Acceso a la SC }
      c[1] := true;
      turno := 1;

      while (c[2] and turno = 1) do
      begin
         null;
      end do

      { Sección crítica }
      c[1] := false;
   end do
end
```

```
Process P2();
begin
   while true do
   begin
      { Acceso a la SC }
      c[2] := true;
      turno := 2;

      while (c[1] and turno = 2) do
      begin
         null;
      end do

      { Sección crítica }
      c[2] := false;
   end do
end
```

Figura 2.8: Algoritmo de Peterson para 2 procesos.

Propiedades de corrección

A continuación, mostramos que la solución de Peterson no solo es una solución para el problema de la exclusión mutua, sino que además garantiza la equidad en el acceso a la misma para los procesos que intervienen en un programa concurrente que usa dicho algoritmo.

Exclusión mutua.

Suponiendo que nos encontramos en el proceso P_i (con $i = 1$ ó $i = 2$), sea j el otro proceso ($j = 1$ si $i = 2$ ó $j = 2$ si $i = 1$), supongamos que el proceso P_i consigue entrar a la sección crítica. Esto sucede solo si $c[j]$ es falso o si `turno` es j. Es decir, si P_j no está intentando acceder a la sección crítica o si P_j fue el último proceso en asignarse el turno.

De esta forma, si el proceso P_j intenta acceder a la vez a la sección crítica (entonces, `c[j] = true`, con lo que `turno = j`), quedará bloqueado, por ser `turno = j` y `c[i] = 1`, con lo que solo un proceso conseguirá al final entrar a la sección crítica.

Alcanzabilidad de la sección crítica.

Supusto que dos procesos intenta acceder a la sección crítica, ambos cambiarán su valor de `c` a `true`, y cambiarán el valor de `turno` a su identificador, quedando uno de ellos bloqueado (el último que modificó dicha variable) y el otro proceso consigue pasar el bucle, con lo que alcanza la sección crítica.

Si solo un proceso P_i intenta acceder a la sección crítica, si notamos por j al identificador del otro proceso, entonces P_i observará `c[j] = false`, con lo que P_i entrará a la sección crítica.

Equidad.

El algoritmo de Peterson no solo garantiza la condición de vivacidad para los procesos del programa concurrente sino que garantiza la equidad en su acceso a la sección crítica:

Supongamos que un proceso P_i está bloqueado esperando a entrar a la sección crítica y un proceso P_j está continuamente entrando y saliendo de la misma monopolizando su uso. Sin embargo, cuando P_j salga de la sección crítica por primera vez, cambiará el valor de `turno` a j, con lo que P_j quedará bloqueado en el acceso a la sección crítica y P_i conseguirá entrar a la misma, contradicción con que un proceso pueda quedar bloqueado.

2.1.7. Algoritmo de Peterson para n procesos

Lo que haremos para obtener el algoritmo de Peterson que resuelve el problema de la exclusión mutua para cualquier número de procesos n será generalizar el caso de dos procesos, suponiendo que ahora tenemos n procesos que quieren entrar a la sección crítica.

Idea

La idea para pasar de 2 procesos a n es repetir $n-1$ veces el proceso de decisión que hicimos anteriormente con 2 procesos.

Es decir, primero llegarán n procesos al protocolo de acceso a la sección crítica y cada uno de estos irá asignando la variable compartida `turno` a su identificador de proceso, de forma que habrá un proceso de todos esos n que será el último en haber cambiado el valor de la variable compartida `turno`. Este proceso será el que se quedará esperando en un bucle y dejaremos a los $n-1$ procesos restantes pasar a realizar este procedimiento otra vez, obteniendo $n-2$ procesos. Repetiremos este procedimiento un total de $n-1$ veces, para obtener un único proceso de todos los que intentaron acceder a la vez a la sección crítica.

A este procedimiento que repetimos para eliminar un proceso del total de procesos le llemaremos *etapa*, de forma que dispondremos de $n-1$ etapas para, en el caso de que lleguen a la vez los n procesos al protocolo de acceso a la sección crítica, dejar pasar solo a uno de ellos a la sección crítica, como resultado de pasar los n procesos por las $n-1$ etapas distintas. Una vez que un proceso haya completado las $n-1$ etapas, se encontrará en la "etapa n", que es la propia sección crítica.

Implementación

Para implementar un número de etapas que coincida con el número de procesos que en un futuro ejecutarán nuestro programa concurrente, nos vemos obligados a crear un bucle que itere $n-1$ veces, de forma que cada iteración del bucle será una nueva etapa. Así mismo, contaremos con una variable j, que será la que nos indique en qué etapa nos encontramos.

Además, necesitaremos de las siguientes variables compartidas:

```
var c : array[0..n-1] of -1..n-2;
    turno : array[0..n-2] of 0..n-1;
```

donde:

- `c` es un array que contiene las claves de los n procesos que ejecutarán nuestro programa concurrente, de forma que `c[i]` será la clave del proceso i-ésimo, la cual podrá tomar los valores:
  - -1, que indica que el proceso está en estado pasivo (no está intentando acceder a la sección crítica).
  - k con $k \in \{0, \ldots, n-1\}$, que indica en qué etapa se encuentra el proceso i-ésimo, entendiendo que la primera etapa es la 0 y la última la $n-2$; siendo la etapa $n-1$ la correspondiente a la sección crítica.
- `turno` es un array de forma que `turno[j]` indica el valor del turno para la etapa j-ésima, cuyo valor es cualquiera de los identificadores de los procesos que ejecutan nuestro programa (como tendremos n procesos, estos están identificados desde el 0 hasta el $n-1$).

Procedemos ahora a mostrar el código del algoritmo de Peterson para n procesos, el cual pasaremos a explicar inmediatamente:

```
Process Pi();
begin
   while true do
   begin
      { - Acceso a la SC - }
      for j=0 to n-2 do
      begin
         c[i] := j;
         turno[j] := i;

         { Existe k <> i con c[k] >= j AND turno[j] = i }
         for k=0 to n-1 do
         begin
            if (k = i) then continue;
            while (c[k] >= j and turno[j] = i ) do
               null;
            end do
         end
      end

      c[i] := n-1;      { Proceso en etapa n-1 (SC) }
```

```
        { - Sección Crítica - }
        c[i] := -1;
    end do
end
```

De esta forma, cuando un proceso llega al protocolo de entrada de la sección crítica, lo que hace es entrar en la primera etapa (`j=0`). En esta, actualiza el valor de su clave a `j` y actualiza la variable `turno` a su identificador.

Posteriormente, ejecuta el `for` indicado en el código, el cual es equivalente a:

```
while(Exists k <> i : c[k] >= j AND turno[j] = i) do
    null;
end do
```

Es decir, si `turno[j] = i` (lo que significa que el proceso i fue el último en cambiar el valor de la variable `turno`) y si hay otro proceso k de forma que `c[k] >= j`, es decir, que existe un proceso en una etapa superior, entonces el proceso i se quedará esperando en la etapa j, hasta que:

- Llegue un nuevo proceso a la etapa j, con lo que el proceso i ya no habrá sido el último en asignar el turno (`turno[j] != i`).
- No haya ningún proceso en ninguna etapa siguiente (es decir, que el proceso que antes había ha terminado ya la ejecución de la sección crítica (ya que es la única forma de avanzar), con lo que no existe un k de forma que `c[k] >= j`).

Ejemplo. Para motivar el buen funcionamiento del algoritmo, el cual demostraremos después, mostramos ya un ejemplo de ejecución del mismo, donde suponemos que tenemos $n = 4$ procesos: P_0, P_1, P_2 y P_3 que quieren acceder a la sección crítica.

Suponiendo que observamos la traza de ejecución:

```
turno[0] = 2; turno[0] = 0; turno[0] = 3; turno[0] = 1;
```

Entonces, obtendríamos el siguiente comportamiento de los procesos (siendo x, y y z valores entre 0 y 3, cuyo valor no nos es relevante):

t	c[0]	c[1]	c[2]	c[3]	turno[0]	turno[1]	turno[2]
0	-1	-1	-1	-1	x	y	z
1	0	0	0	0	2	y	z
2	0	0	1	0	0	2	z
3	1	0	2	0	3	0	2
4	2	0	3	1	1	3	0
5	3	1	-1	2	1	1	3
6	-1	2	-1	3	1	1	1
7	-1	3	-1	-1	1	1	1
7	-1	-1	-1	-1	1	1	1

Tabla 2.1: Evoluión de los estados de los procesos.

Donde hemos supuesto que todos los procesos llevan una velocidad de ejecución similar, de forma que en cada unidad de tiempo, un proceso es capaz de cambiar

su clave, asignar su turno y comprobar la condición del bucle `while`. De esta forma (cada índice indica el valor de t):

(0) Ningún proceso accede al protocolo de entrada a la sección crítica.

(1) Todos los procesos acceden a la vez al protocolo de entrada, a la etapa 0, de forma que P_2 es el primero en asignar el valor de `turno[0]`.

(2) Como no hay ningún proceso en la etapa 1 (no existe k de forma que `c[k] >= j`), entonces el proceso P_2 pasa a la siguiente etapa, la 1. En este mismo instante, P_0 cambia el valor de `turno[0]`.

(3) Ahora, P_3 cambia el valor de `turno[0]`, lo que provoca que P_2 no haya sido el último proceso en cambiar el turno en la etapa 0, con lo que avanza a la etapa 1, cambiando su valor, lo que hace que P_2 ya no haya sido el último, avanza a la etapa 2.

(4) P_2 ve que no hay ningún proceso en la etapa 3 (la sección crítica), con lo que avanza a dicha etapa. En este momento, P_1 cambia el valor de `turno[0]`, con lo que P_3 avanza a la siguiente etapa, lo que provoca que P_0 también avance de etapa.

(5) P_2 abandona la sección crítica, pasando ahora a estar pasivo, lo que hace que P_0 pueda entrar en la sección crítica.

Notemos que lo que hemos hecho ha sido elegir una traza muy concreta para mostrar este ejemplo de ejecución, en el que hemos intentado dar ejemplos de las dos razones por las que un proceso puede avanzar a la siguiente etapa:

- Por no haber ningún proceso en ninguna etapa siguiente.
- Por no ser el último proceso en haber cambiado la variable `turno` en la etapa actual.

Propiedades de corrección

A continuación, mostraremos las propiedades de corrección del algoritmo de Peterson para n procesos. Para ello, será necesario antes introducir una definición y varios lemas:

Definición 2.1 (Precedencia)**.** Dados dos procesos, P_i y P_k, decimos que P_i precede a P_k si $c[i] > c[k]$. Es decir, si P_i se encuentra en un estado posterior al estado de P_k.

Además, será común nombrar "condición de espera de la etapa j", con lo que nos estaremos refiriendo a la condición:

```
Exists k <> i : c[k] >= j AND turno[j] = i.
```

Lema 2.1. *Un proceso que precede a todos los demás puede avanzar al menos una etapa.*

Demostración. Sea P_i un proceso que precede a todos los demás, entonces tendremos que $c[i] = j > c[k]\ \forall k \neq i$. De esta forma, no existirá ningún proceso en una etapa superior, con lo que la condición de espera en la etapa j será falsa, permitiendo a dicho proceso avanzar a la etapa $j + 1$.

Sin embargo, como la evaluación de la condición de espera en una etapa no se realiza de forma atómica (ya que debemos implementar dicha condición mediante un bucle `for` que itere sobre k), puede ocurrir que uno o varios procesos alcancen también la etapa j en la que se encuentra P_i mientras comprueba la condición de espera de la etapa j. Sin embargo, tan pronto como esto suceda, algún proceso cambiará el valor de la variable `turno[j]`, lo que hará que el proceso P_i evalúe la condición de espera de la etapa j como falsa, lo que le permita avanzar a la etapa $j + 1$. □

Lema 2.2. *Cuando un proceso pasa de la etapa j a la $j+1$, se ha de verificar alguna de las dos siguientes condiciones:*

a) Que dicho proceso preceda a todos los demás.

b) Que dicho proceso no estuviese solo en la etapa j.

Demostración. Supuesto que el proceso P_i está a punto de pasar de la etapa j a la $j + 1$, entonces la condición de espera de la etapa j ha de ser falsa para P_i, con lo que:

- No existe $k \neq i$ que verifique $c[k] \geqslant j$, es decir, que $c[k] < j = c[i]$ para todo $k \neq i$, con lo que P_i precede a todos los demás procesos.
- $turno[j] \neq i$, con lo que existirá otro proceso P_k de forma que haya sido el último en modificar el valor de $turno[j]$, con lo que dicho proceso P_k ha de estar necesariamente en la etapa j, al igual que P_i.

□

Lema 2.3. *Si tenemos al menos dos procesos en la etapa j, entonces ha de haber, al menos, un proceso en cada etapa anterior.*

Demostración. Distinguimos dos casos:

$j = 1$. Supuesto que tenemos dos procesos en la etapa 1, P_i y P_k, entonces alguno de ellos habrá sido el último en pasar de la etapa 0 a la 1. No perdemos generalidad suponiendo que dicho proceso es P_i. En dicho momento, y por el Lema 2.2, se cumplirá que o bien P_i precede a todos los demás o bien que no estaba solo en la etapa 0.

Sin embargo, como P_k pasó antes a la etapa 1 que P_i, es imposible que P_i preceda a todos los demás al pasar a la etapa 1, por no preceder a P_k. Por ello, P_i no estaba solo cunado pasó de la etapa 0 a la 1, con lo que habrá otro proceso, P_l, en la etapa 0.

De esta forma, tenemos demostrado el lema para el caso $j = 1$.

Supuesto que el lema es cierto para $j-1$, veámosolo para j. Por un razonamiento análogo al anterior, supuesto que tenemos dos procesos, P_i y P_k en la etapa j, y supuesto que P_i fue el último de ellos en avanzar desde la etapa $j-1$ a la j, por el Lema 2.2, dicho proceso no estaba solo en la etapa $j-1$, con lo que hay al menos un proceso en cada etapa, desde la 0 hasta la $j-1$.

□

Lema 2.4. *El número máximo de procesos que puede haber en la etapa j es de $n-j$ procesos.*

Demostración. Supuesto que tenemos al menos dos procesos en la etapa j, entonces por el Lema 2.3 las etapas 0, 1, ..., $j-1$ (son j etapas) han de contener al menos un proceso, con lo que ninguno de estos procesos puede estar en la etapa j, con lo que el número máximo de procesos en la etapa j es $n-j$. □

Estamos ya listos para probar todas las propiedades de corrección del algoritmo de Peterson:

Exclusión mutua.

- Supuesto que tenemos un proceso en la etapa $n-2$ y uno en la $n-1$ (en la sección crítica), entonces el proceso de la etapa $n-2$ no puede avanzar, ya que este proceso no precede a todos los demás y está solo en dicha etapa, con lo que de avanzar, no se cumpliría el Lema 2.2.

  Sin embargo, cuando el proceso de la etapa $n-1$ abandone dicha etapa, entonces el proceso de la etapa $n-2$ se encontraría en las hipótesis del Lema 2.1, con lo que podría avanzar a dicha etapa.
- Supuesto que tenemos dos procesos en la etapa $n-2$, en cuanto uno avance, el otro no podrá hacerlo, tal y como acabamos de razonar.
- Gracias al Lema 2.2, sabemos que el número máximo de procesos en la etapa $n-2$ es de 2, por lo que la distinción de casos está ya completa.

Alcanzabilidad de la sección crítica.

Por reducción al absurdo, supongamos que hay una traza de ejecución del programa concurrente en la que la sección crítica es inalcanzable, es decir, que ningún proceso puede llegar a la sección crítica. En dicho caso, suponemos que tenemos un proceso en una etapa j y distingamos casos:

- Si dicho proceso precede a todos los demás, entonces por el Lema 2.1 este puede avanzar.
- Si no, hay al menos un proceso que le precede, es decir, que está en una etapa superior a j:
  - Si dicho proceso o cualquier otro precede a todos los demás, por el Lema 2.1, este puede avanzar.
  - Si no hay ningún proceso que preceda a todos los demás, entonces hay al menos dos procesos en la etapa más avanzada, con lo que alguno fue el último en asignar el valor de la variable `turno` de dicha etapa, por lo que para el resto de procesos la condición de espera de dicha etapa será falsa, permitiéndoles avanzar.

En cualquier caso, habrá algún proceso que consiga avanzar en cualquier etapa, con lo que dicha situación es imposible.

Equidad

No solo demostraremos la vivacidad del algoritmo de Peterson, sino también la equidad de los procesos en el acceso a la sección crítica. Para ello, supongamos el caso más desfavorable en la entrada a la sección crítica:

Es decir, que los n procesos intentan a la vez entrar a la sección crítica, de forma que el proceso P_i es el último de los n procesos en modificar el valor de la variable `turno[0]`, con lo que se bloqueará en dicha etapa y dejará pasar al resto de los $n-1$ procesos. P_i permanecerá bloqueado en esta etapa hasta que:

- Preceda a todos los demás procesos, con lo que todos los $n-1$ procesos anteriores hayan ejecutado ya la sección crítica.
- Venga un nuevo proceso a la etapa 0, es decir, haya un proceso P_k que ya haya ejecutado la sección crítica y entre de nuevo en el protocolo de entrada, lo que permitirá a P_i avanzar, al menos en una etapa.

Dicho procedimiento se repetirá, hasta la etapa $n-2$, donde el proceso P_i conseguirá al fin entrar en la sección crítica.

De esta forma, si $r(n)$ es el número de turnos de espera para n procesos, entonces tenemos que en el peor caso, el proceso P_i deberá dejar pasar a $n-1$ procesos (con lo que pasarán $n-1$ turnos), así como esperar los turnos correspondientes a la misma situación pero para $n-1$ procesos. Es decir:

$$r(n) = n - 1 + r(n-1)$$

Por tanto, el número de turnos máximos de espera para entrar en la sección crítica según el algoritmo de Peterson está limitado por el valor

$$r(n) = \frac{n(n-1)}{2}$$

Recordando el algoritmo de Knuth, este era un algoritmo más complicado que también resolvía el problema de la exclusión mutua. Sin embargo, pese a la sencillez del algoritmo de Peterson, es más eficiente en cuanto al acceso a la sección crítica, ya que el orden del tiempo de espera en el algoritmo de Knuth era del orden $O(2^n)$, siendo este $O(n^2)$ en el algoritmo de Peterson.

Por tanto, decimos que el algoritmo de Peterson es la solución canónica al problema de la exclusión mutua, al ser el algoritmo más eficiente de los dos estudiados en esta Sección que cumplen con la quinta propiedad de corrección (la de demostrar un retraso máximo en el acceso de los procesos a la sección crítica).

2.2. Definición de un monitor

El concepto de semáforo se desarrolló previamente en el Seminario 1 de prácticas[3]. Los semáforos presentan dos grandes limitaciones:

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

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.

La idea básica de monitor es un módulo que contiene un conjunto de variables a las que llamaremos *variables permanentes*[4], 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 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.

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

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.

2.2.1. 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[5] 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 de forma intuitiva en la Tabla 2.2, como un conjunto que englobla:

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

- 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 2.2: 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[6].

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):

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

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

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

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

```
  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 = atras) 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 `retirar`, 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[7].

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 de tipo `cond` por cada razón por la que queremos bloquear un proceso[8].

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**.

2.2.2. 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 solo 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

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

[8]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.

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[9]). 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 2.2. 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 locales tras desbloquearse. Es decir, el proceso no debe enterarse localmente[10] 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.

Copias paramétricas de un monitor

El siguiente ejemplo nos ilustra cómo podemos crear nuevos monitores a partir de uno ya creado, fijando parámetros que use el código de inicialización.

Ejemplo. Aunque los monitores están pensados 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 solo nos interesa la forma en la que los monitores inicializan sus variables permanentes[11].

[9] Se explicará más adelante.

[10] Las variables locales a la función deben mantenerse, pero puede haber variables globales que sí hayan cambiado.

[11] 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.

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:

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

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

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

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

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

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

2.2.3. Exclusión mutua en los procedimientos de un monitor

Si tenemos varios procesos del programa concurrente que quieren hacer uso de procedimientos del monitor a la vez, solo podremos dejar pasar un proceso al monitor (suponiendo que este se encuentre libre). Para los otros procesos, almacenaremos 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, solo 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.

Observación. 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:

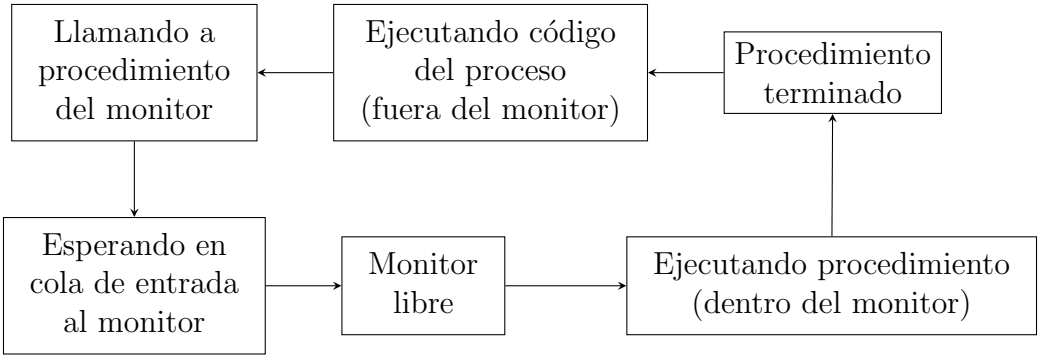


Figura 2.9: Vida de un proceso en un programa concurrente con monitores.

De esta forma, podemos ahora reescribir la descripción gráfica de monitor que hicimos en la tabla 2.2, incluyendo ahora la cola de entrada al monitor, tal y como vemos en la tabla 2.3

Cola del monitor
Variables permanentes
Procedimientos exportados
Código de inicialización

Tabla 2.3: Esquema de un monitor incluyendo la cola de entrada.

2.2.4. 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` 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 queramos bloquear a los procesos concurrentes de nuestro programa por alguna determinada razón, hasta que se cumpla una condición determinada, a la que llamaremos *condición de sincronización*. 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 solo 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[12]. Si esta cola está vacía, es equivalente a una operación nula[13].

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[14]. Se recomienda **no usarla**.

[12] Hemos de tener cuidado con esto, se explicará más adelante.

[13] Esto es, equivalente a la instrucción `;`.

[14] Se explicará más adelante qué es esto.

De esta forma, la representación gráfica final de un monitor es la que se muestra en la tabla 2.4:

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

Tabla 2.4: Esquema de un monitor incluyendo las variables condición.

Semánticas de señales

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` sobre una variable condición (recordemos que tenemos al menos un proceso bloqueado), desbloqueará al primer proceso de su cola asociada, que estaba ejecutando código del monitor, por lo que ahora tenemos dos procesos ejecutando código del monitor: el proceso que señala y el señalado (el recién desbloqueado). Esta condición no puede darse, ya que los procedimientos de un monitor deben ejecutarse en exclusión mutua. Una solución a este problema es que el procedimiento que señala se bloquee en la cola de entrada al monitor[15], dejando paso al recién desbloqueado.

Esta solución plantea un problema, y es que si el proceso señalador se bloquea, puede que en algún momento deje al monitor libre, por lo que se meta un proceso de la cola de entrada al monitor, planteando nuevamente la situación en la que tenemos dos procesos en el monitor (el desbloqueado y el primero que estaba esperando entrar al monitor). Deberá haber un mecanismo que elija quién de los dos acaba finalmente con el monitor. En caso de que sea el proceso que estaba esperando en la cola de entrada, diremos que se produce un *robo de señal*, donde el proceso recién desbloqueado debe irse al final de la cola de entrada al monitor.

[15] Veremos más soluciones.

Para solucionar este segundo problema, algunos lenguajes implementan una *semántica desplazante*[16] 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. En esta sección y en la siguiente, supondremos que estaremos trabajando siempre con señales `signal` con semántica desplazante, de forma que el proceso señalador se bloqueará tras ejecutar un `signal` y podemos pensar que es enviado a la cola de entrada al monitor. En una futura sección veremos los tipos de semánticas de señales que podemos encontrarnos (cada uno hará que el comportamiento de `signal` sea distinto).

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` necesarias. Para comprobar esto, usaremos nuestro sistema de verificación formal.

2.3. 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.

[16] Se trabajará más adelante sobre las semánticas de las señales.

2.3.1. Invariante de monitor

Definición 2.3 (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[17], 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**:

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

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.

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

Cuidado con las intereferencias

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):

```
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:

```
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.

2.3.2. Axiomas para operaciones de sincronización desplazantes

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[18].

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 locales del procedimiento del monitor) antes y después[19] 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 solo 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:

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

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

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

[19] Gracias a que estamos en lenguajes reentrantes.

Observación. Notemos que el axioma de la operación `signal` funciona porque hemos suspuesto que **tenemos semántica desplazante**, ya que después de ejecutar `signal` desbloqueamos al proceso que teníamos bloqueado, cediéndole el monitor, por lo que dicho proceso seguirá procesando su procedimiento. Cuando el proceso señalador vuelva al monitor, solo podremos garantizar que se cumple IM, ya que tanto el proceso señalado como cualquier otro que se introduzca en el monitor después del señalado (veremos más adelante si esto es posible), pueden cambiar la condición C, por lo que solo podemos esperar IM.

Una vez vistos ya todos los axiomas sobre verificación de operaciones de sincronización de monitores, 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:

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

a) Comenzamos por el segundo, por ser más sencillo. Tenemos:

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

Por tanto, 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`:

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

Por tanto, por el axioma de la operación `wait` usado con $L = \{V\}$, tenemos que el siguiente triple es cierto:

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

Como $\{IM \wedge s = 0\} \rightarrow \{IM\}$, por la segunda regla de la consecuencia, tenemos demostrado el triple que buscábamos.

Una vez demostrados los dos triples, tenemos probado el triple del `if`, por lo que solo 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. Se pide demostrar que el siguiente monitor funciona como un semáforo de Habermann. Un semáforo de Habermann se trata de un semáforo normal que lleva la cuenta de:

- El número de llamadas realizadas a `signal`, `nv`.
- El número de llamadas realizadas a `wait`, `na`.
- El número de llamadas completadas a `wait`, `np`.

En este caso, llamaremos `P` al procedimiento que simule la operación `wait` y `V` al procedimiento que simule la operación `signal`.

```
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
```

Buscamos un IM para preceder a la demostración del mismo. Notemos que las variables permanentes han de cumplir:

- $np \leqslant na$, ya que para completar una llamada `P` hay que realizar una llamada.
- $np \leqslant nv$, ya que, como inicialmente $np = na = nv = 0$, para poder completar una llamada a `P`, hay que previamente haber hecho una llamada a `V`.
- $np \geqslant \text{mín}(na, nv)$, para no detener innecesariamente a los preocesos, cumpliendo la hipótesis de progreso finito.

De estas tres propiedades, deducimos que el invariante a usar es:

$$\{IM\} \equiv \{np = \text{mín}(na, nv)\}$$

Pasemos ahora a la demostración del monitor:

1. En primer lugar, probaremos el triple de inicialización de variables:

$$\{V\}$$
$$na = 0;\ np = 0;\ nv = 0;$$
$$\{na = 0 \land np = 0 \land nv = 0\} \rightarrow \{IM\}$$

2. Posteriormente, probaremos el triple del procedimiento `P`:

$$\{IM \land L\}\ P\ \{C \land L\}$$

   para ello:

$$\{IM\} \equiv \{np = \text{mín}(na, nv)\}$$
$$na = na + 1;$$
$$\{np = \text{mín}(na - 1, nv)\}$$
$$\texttt{if}\ \ (na > nv)\ \ \texttt{then}$$
$$\{na > nv \land np = \text{mín}(na - 1, nv)\} \rightarrow \{na - 1 \geqslant nv \land np = \text{mín}(na - 1, nv)\} \rightarrow$$
$$\rightarrow \{na > nv \land np = nv\} \rightarrow \{np = \text{mín}(na, nv)\}$$
$$c.wait();$$

   Donde en la precondición de la operación `c.wait();` hemos necesitado comprobar que IM se seguía compliendo.

Y para poder seguir, hemos de buscar la precondición de la operación `c.signal();` asociada a la misma variable condición, con lo que comenzamos a demostrar el procedimiento `V`:

$$\{IM\} \equiv \{np = \text{mín}(na, nv)\}$$
$$nv = nv + 1;$$
$$\{np = \text{mín}(na, nv - 1)\}$$
$$\texttt{if } (na > np) \texttt{ then}$$
$$\{na > np \wedge np = \text{mín}(na, nv - 1)\} \rightarrow \{na > np \wedge np = nv - 1\}$$

Luego tenemos ya la poscondición de la operación `c.wait();`, con lo que podemos volver por donde íbamos:

$$\{IM\} \equiv \{np = \text{mín}(na, nv)\}$$
$$na = na + 1;$$
$$\{np = \text{mín}(na - 1, nv)\}$$
$$\texttt{if } (na > nv) \texttt{ then}$$
$$\{na > nv \wedge np = \text{mín}(na - 1, nv)\} \rightarrow \{na - 1 \geqslant nv \wedge np = \text{mín}(na - 1, nv)\} \rightarrow$$
$$\rightarrow \{np = nv\} \rightarrow \{np = \text{mín}(na, nv)\}$$
$$c.wait();$$
$$\{na > np \wedge np = nv - 1\}$$
$$\texttt{else do}$$
$$\{na \leqslant nv \wedge np = \text{mín}(na - 1, nv)\} \rightarrow \{na - 1 < nv \wedge np = \text{mín}(na - 1, nv)\} \rightarrow$$
$$\rightarrow \{na - 1 < nv \wedge np = na - 1\} \rightarrow \{np < nv \wedge np = na - 1\}$$
$$\texttt{null;}$$
$$\{np < nv \wedge np = na - 1\}$$
$$\texttt{endif}$$

Como las poscondiciones de cada bloque del `if` son distintas, debemos relajarlas para buscar dos poscondiciones iguales, para poder aplicar la regla del `if`. Notemos que:

$$\{na > np \wedge np = nv - 1\} \rightarrow \{na > nv - 1 \wedge np = nv - 1\} \rightarrow$$
$$\rightarrow \{na \geqslant nv \wedge np = nv - 1\}$$
$$\{np < nv \wedge np = na - 1\} \rightarrow \{na - 1 < nv \wedge np = na - 1\} \rightarrow$$
$$\rightarrow \{na \leqslant nv \wedge np = na - 1\}$$

Podemos por tanto, relajar ambas poscondiciones a la poscondición

$$\{np + 1 = \text{mín}(na, nv)\}$$

Con lo que ahora sí podemos aplicar la regla del `if`, con lo que podemos

finalizar la demostración del triple del procedimiento `P`:

$$\{IM\} \equiv \{np = \text{mín}(na, nv)\}$$
$$na = na + 1;$$
$$\{np = \text{mín}(na - 1, nv)\}$$
$$\texttt{if } (na > nv) \texttt{ then}$$
$$\{np = \text{mín}(na, nv)\}$$
$$c.wait();$$
$$\{np + 1 = \text{mín}(na, nv)\}$$
$$\texttt{else do}$$
$$\{np + 1 = \text{mín}(na, nv)\}$$
$$\texttt{null;}$$
$$\{np + 1 = \text{mín}(na, nv)\}$$
$$\texttt{endif}$$
$$\{np + 1 = \text{mín}(na, nv)\}$$
$$np = np + 1;$$
$$\{np = \text{mín}(na, nv)\} \equiv \{IM\}$$

3. Para probar ahora el triple del procedimiento `V`:

$$\{IM\} \equiv \{np = \text{mín}(na, nv)\}$$
$$nv = nv + 1;$$
$$\{np = \text{mín}(na, nv - 1)\}$$
$$\texttt{if } (na > np) \texttt{ then}$$
$$\{na > np \wedge np = \text{mín}(na, nv - 1)\} \rightarrow \{na > np \wedge np = nv - 1\}$$
$$c.signal();$$
$$\{np = \text{mín}(na, nv)\}$$
$$\texttt{else do}$$
$$\{na \leqslant np \wedge np = \text{mín}(na, nv - 1)\} \rightarrow \{np = na\} \rightarrow \{np = \text{mín}(na, nv)\}$$
$$\texttt{null;}$$
$$\{np = \text{mín}(na, nv)\}$$
$$\texttt{endif}$$
$$\{np = \text{mín}(na, nv)\} \equiv \{IM\}$$

Con lo que tenemos provado que IM es un IM.

2.3.3. Regla de concurrencia para programas con monitores

Consideramos un programa concurrente en el que tenemos n procesos ejecutándose que podemos representar como triples 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.

2.4. Patrones de uso de un monitor

Al programar programas concurrentes que nos resuelvan un problema, encontramos muchas veces ciertos pequeños problemas a resolver que se repiten a menudo. En esta sección, destacamos tres de estos problemas, describiéndolos, planteando una solución mediante un monitor a utilizar en ellos y demostrando que el monitor realiza el funcionamiento esperado.

2.4.1. Espera única

Puede suceder que en un programa concurrente queramos que un proceso espere a que otro proceso haya realizado una cierta acción para seguir con su ejecución. Llamaremos al proceso que tiene que esperar al otro *consumidor* y a dicho otro *productor*.

El problema se resuelve de forma muy sencilla, con una variable compartida que indique si el productor ha realizado ya su acción por la que el consumidor debe esperar o si no lo ha hecho todavía. Cuando el consumidor se acerque a la zona en la que debe esperar al productor, consultará la variable compartida y en caso de que esta indique que no se ha realizado la acción, bloquearemos al proceso. Si se ha realizado la acción, no haremos nada.

Además, cuando el productor haya realizado la acción que ha de realizar, cambiaremos el valor de dicha variable compartida, indicando que ya se ha realizado la acción. En caso de que el consumidor se haya bloqueado antes de realizar la acción, lo desbloquearemos.

Observemos que estamos haciendo uso de una variable compartida, que es modificada por un proceso. Debemos por tanto acceder a ella en exclusión mutua. Esto es garantizado por el uso del monitor.

Monitor a usar

El monitor que usaremos tendrá dos procedimentos exportables, uno llamado `esperar` que será el que use el proceso consumidor, y otro llamado `notificar`, que será el que use el proceso productor para avisar al consumidor de que ya ha realizado la acción.

De esta forma, podemos ver el código monitor en pseudo código:

```
monitor EU;
var terminado : boolean;  {si terminado = true, se ha realizado la acción}

    {variable auxiliar para la demostracion}
    autorizado : boolean; {si autorizado = true, consumidor puede ejecutarse}

    c : cond;

   procedure esperar(); begin
      if (not terminado) then
         c.wait();
      autorizado = true;
   end

   procedure notificar(); begin
      terminado = true;
      c.signal();
   end

   begin {codigo de inicializacion}
      terminado = false; autorizado = false;
   end
```

Para su demostración, usaremos el invariante

$$\{IM\} \equiv \{terminado = false \Longrightarrow autorizado = false\}$$

La demostración se deja como ejercicio al lector.

2.4.2. Exclusión mutua

Es muy habitual que en programas concurrentes tengamos una o varias regiones de código que queramos que se ejecuten en exclusión mutua, llamadas secciones críticas. Es decir, mientras que un proceso se encuentre ejecutando una sección crítica, ningún otro proceso podrá estar ejecutando a la vez la misma sección crítica[20].

Usualmente, querremos tener exclusiones mutuas cuando varios procesos de un programa hagan uso de un recurso compartido, tal como una variable compartida, una salida a un fichero o imprimir información en un entorno gráfico.

Monitor a usar

El monitor que resuelve el problema de la exclusión mutua tendrá dos procedimientos exportables, `entrar`, que se ejecutará antes de cualquier sección crítica, y `salir`, que se ejecutará al final de cada sección crítica.

De esta forma, tenemos el monitor:

[20] Podemos tener dos secciones críticas con distintos códigos pero que sean referidas al acceso para la misma variable compartida. En dicho caso, pensamos que las secciones críticas son iguales, ya que solo puede haber un proceso que ejecute una u otra a la vez.

```
monitor EM;
var ocupada : boolean;  {ocuapda = true si hay un proceso en seccion critica}
    cola : cond;

   procedure entrar(); begin
      if ocupada then
         cola.wait();
      ocupada = true;
   end

   procedure salir(); begin
      ocupada = false;
      cola.signal();
   end

   begin
      ocupada = false;
   end
```

Para demostrarlo, primero definiremos la variable num_{sc} como el número de procesos que se encuentran ejecutando la sección crítica. Una vez definido, el invariante a usar será

$$\{IM\} \equiv \{(ocupada = false \iff num_{sc} = 0) \land 0 \leqslant num_{sc} \leqslant 1\}$$

La demostración se deja como ejercicio al lector.

2.4.3. Productores/Consumidores

Volvemos otra vez al paradigma del productor/consumidor, el cual hemos explicado varias veces ya en este documento. Ahora, admitiremos la existencia de varios procesos productores que querrán generar datos y escribirlos en una variable compartida, así como varios consumidores, que querrán leer dicha variable compartida y realizar los cálculos pertinentes.

Monitor a usar

Usaremos un monitor con dos procedimientos: `escribir`, que permitirá escribir en la variable compartida el valor indicado, y `leer`, que permitirá leer el valor de la variable compartida. La ventaja de usar un monitor es que los códigos de sincronización solo los tenemos que realizar en el monitor, dejando limpios los códigos de los procesos.

Como tenemos dos condiciones de sincronización, bloquear a productores que quieran escribir en una variable que no se ha leído, o bloquear a consumidores que quieran leer de una variable cuyo valor ya ha sido leído, necesitaremos dos variables de tipo `cond`:

```
monitor PC;
var valor : integer;  {variable compartida a usar}
    pendiente : boolean;  {si pendiente = true, valor escrito y no leido}
    cola_prod, cola_cons : cond;

   procedure escribir(v : integer); begin
```

```
      if pendiente then
         cola_prod.wait();
      valor = v;
      pendiente = true;
      cola_cons.signal();
   end

   procedure leer() : integer; begin
      if (not pendiente) then
         cola_cons.wait();
      result = valor;
      pendiente = false;
      cola_prod.signal();
   end

   begin
      pendiente = false;
   end
```

Para la demostración, debemos definir primero:

$$E = \text{ número de llamadas a escribir } \textbf{completadas}.$$
$$L = \text{ número de llamadas a leer } \textbf{completadas}.$$

De esta forma, podemos definir el invariante

$$\{IM\} \equiv \left\{ E - L = \left\{ \begin{array}{ll} 0 & \text{si } pendiente = false \\ 1 & \text{si } pendiente = true \end{array} \right\} \right\} \equiv$$
$$\equiv \{(pendiente = false \wedge E - L = 0) \vee (pendiente = true \wedge E - L = 1)\}$$

La demostración se deja como ejercicio para el lector.

Asímismo, notemos que la demostración es similar a la del monitor para exclusión mutua, teniendo en cuenta que:

$$E - L = num_{sc}$$
$$pendiente = \neg libre$$

Ejercicio. Si ahora los productores no escriben sobre una misma variable compartida sino sobre un buffer (por ejemplo, un array con planificación FIFO), plantear un monitor que solucione el problema de los productores/consumidores, así como desmostrar que dicho monitor funciona correctamente.
(**Pista:** para la demostración, sustituir en el IM "1" por el tamaño del buffer)

2.5. Semánticas de señales

Como comentamos ya al inicio del Capítulo, las operaciones `signal` de las variables condición son operaciones delicadas, ya que son ejecutadas por un proceso que se encuentra ejecutando algún procedimiento del monitor y que lo que hacen es desbloquear a algún proceso se que se encontraba ejecutando código del monitor,

por lo que (si no hacemos nada), tendremos dos procesos distintos ejecutando a la vez procedimientos (podría ser el mismo procedimiento) de un mismo monitor, algo que no puede darse.

Para solucionar el problema que nos plantea la operación `signal`, plantearemos distintas *semánticas de señales* `signal`. Esto es, plantearemos varios paradigmas en los que la señal `signal` tendrá un significado u otro, de forma que su finalidad sea siempre *sacar al primero proceso de la cola de la variable condición* en cuestión y ponerlo en otro sitio que permita que dicho proceso entre al monitor en algún futuro próximo, garantizando la propiedad de vivacidad de que dicho proceso en algún momento volverá a entrar al monitor.

Además, clasificaremos los distintos tipos de semánticas de señales que veremos en **no desplazantes** y **desplazantes**.

Definición 2.4. Diremos que **una semántica de señal es desplazante** si, siempre que un proceso ejecute una operación `signal` sobre una variable condición `c`, en dicho instante cederá el monitor al primer proceso de la cola de bloqueados de la variable condición `c` sin que el monitor quede libre en ningún momento[21]. Además, debe garantizarse la propiedad de vivacidad de que el proceso señalador volverá a entrar al monitor en algún momento.

Veremos ahora todas las posibles semánticas de señales que podemos encontrarnos, entendiendo que el proceso *señalador* es aquel que ejecuta la operación `signal` sobre una variable condición; y que el proceso *señalado* es aquel que estaba primero en la cola de bloqueados de la misma variable condición.

2.5.1. Señalar y Continuar (SC)

El proceso señalador no se bloquea tras ejecutar `signal`, sino que sigue con la ejecución del procedimiento en cuestión. El proceso señalado se bloquea hasta que se pueda adquirir de nuevo el monitor.

Como podemos ver, se trata de una semántica no desplazante, ya que el proceso señalador no se bloquea tras ejecutar la instrucción `signal`. En relación al proceso señalado, se produce una *competición* contra el resto de procesos que esperan en la cola de entrada al monitor. Esta "competición" que hemos mencionado depende de la implementación que se haga, destacando dos posibilidades:

1. El proceso señalado pasa al final de la cola de entrada al monitor.

2. Después de que el proceso señalador deje libre el monitor (ya sea porque termina el procedimiento o se bloquea debido a una instrucción `wait`), se desbloquea al proceso señalado y al primero de la cola de entrada al monitor, se sucede una condición de carrera entre ambos y el vencedor (el que primero llega al monitor), acaba ejecutándolo. El perdedor acaba al final de la cola de entrada al monitor.

[21] Evitando que entre al monitor cualquier otro proceos de la cola de entrada al monitor.

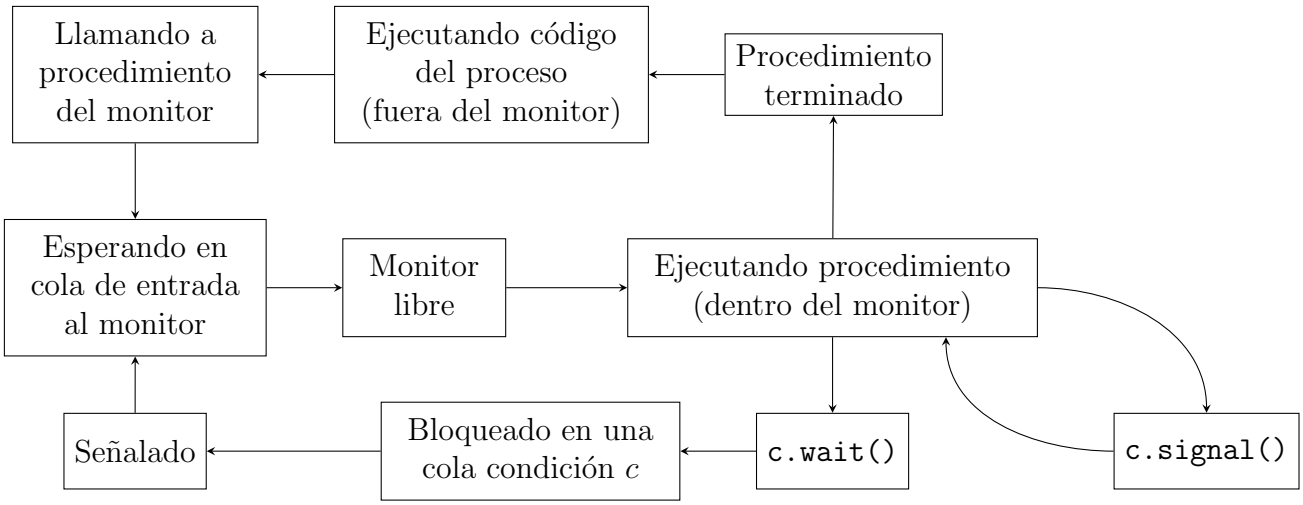


Figura 2.10: Vida de un proceso en un programa con monitores con semántica SC.

> Como podemos entender, la primera opción es mejor, ya que en la segunda puede suceder que un proceso pierda varias veces la competición por el monitor, incumpliendo la condición de equidad entre procesos; además de que no sabemos qué sucede tras desbloquear a dicho proceso, debido a que la competición es transparente para el programador.

De esta forma, podemos representar la vida de un proceso que forma parte de un programa concurrente que usa monitores con semántica de señales de señalar y continuar con el diagrama de la Figura 2.10.

A tener en cuenta para bloquear

Como se trata de una semántica no desplazante, debemos tener cuidado con un detalle:

En semántica desplazante, bloqueamos a un proceso porque se acerca a un estado inseguro del programa. Cuando nos alejemos de dicho estado, realizaremos la instrucción `signal`, por lo que el proceso que bloqueamos se podrá ejecutar al habernos alejado del estado inseguro del programa.

Por otra parte, ahora también desbloquearemos al proceso por alejarnos de dicho estado inseguro, pero el proceso señalado no se ejecutará tras esto, por lo que puede que se ejecuten procesos en medio que vuelvan a acercarse a este estado inseguro, luego tendremos que volver a comprobar si estamos cerca o no de dicho estado inseguro.

Por tanto, en semánticas se señales no desplazantes, en vez de bloquear a procesos con:

```
1 if (cerca de estado inseguro) then
    c.wait();
```

Tendremos que plantear el código

```
1 while (cerca de estado inseguro) do begin
    c.wait();
  end
```

Ya que cuando el proceso señalado vuelva no sabremos si la condición por la que lo bloqueamos es cierta o no. Puede suceder que bloqueemos a un proceso por acercarse a un estado inseguro, que lo desbloqueemos cuando el estado del programa se aleje de dicho estado, que se ejecuten procesos que se acerquen a dicho estado y que cuando el proceso que fue señalado entre al monitor, se encuentre cerca del mismo estado inseguro por el que tuvo que bloquearse, teniendo que hacerlo nuevamente.

2.5.2. Señalar y Salir (SS)

El proceso señalador **finaliza** la ejecución de su procedimiento tras la ejecución de una instrucción `signal`. El proceso señalado se desbloquea posteriormente y en todo este tiempo el monitor no queda libre en ningún momento. Se trata, por tanto, de una semántica desplazante, ya que el proceso señalador cede el monitor al proceso señalado.

Notemos que en señalar y salir, el proceso señalador **finaliza** la ejecución de su procedimiento. Es decir, cualquier instrucción que haya tras una operación `signal` no se ejecutará nunca. Obliga por tanto a una disciplina de programación en la que si queremos realizar un `signal` dentro de un procedimiento, esta instrucción debe ser la última dentro del procedimiento, para poder ejecutar antes todas las instrucciones que deseemos.

Si recordamos el ejemplo de la Sección 2.2.1, observamos que en este las instrucciones `signal` se encuentran al final de los procedimientos de forma natural. Por tanto, la semántica SS funciona bien para este monitor.

Al igual que hicimos para SC, podemos representar la vida de un proceso que forma parte de un programa concurrente que usa monitores con semántica de señales de señalar y salir con el diagrama de la Figura 2.11.

2.5.3. Señalar y Esperar (SE)

El proceso señalador se bloquea al final de la cola de entrada al monitor y cede el monitor al proceso señalado. El monitor no queda libre en ningún momento. Se trata de otro tipo de señal con semántica desplazante.

Puede considerarse una semántica *injusta*, ya que cada vez que un proceso ejecuta la operación `signal`, debe irse al final de la cola de entrada al monitor, teniendo que esperar entre todos los procesos nuevamente (el proceso probablemente tuvo ya que esperar para entrar al monitor para ejecutar el procedimiento que contenía la operación `signal`).

Podemos representar la vida de un proceso que forma parte de un programa concurrente que usa monitores con semántica de señales de señalar y esperar con el diagrama de la Figura 2.12.

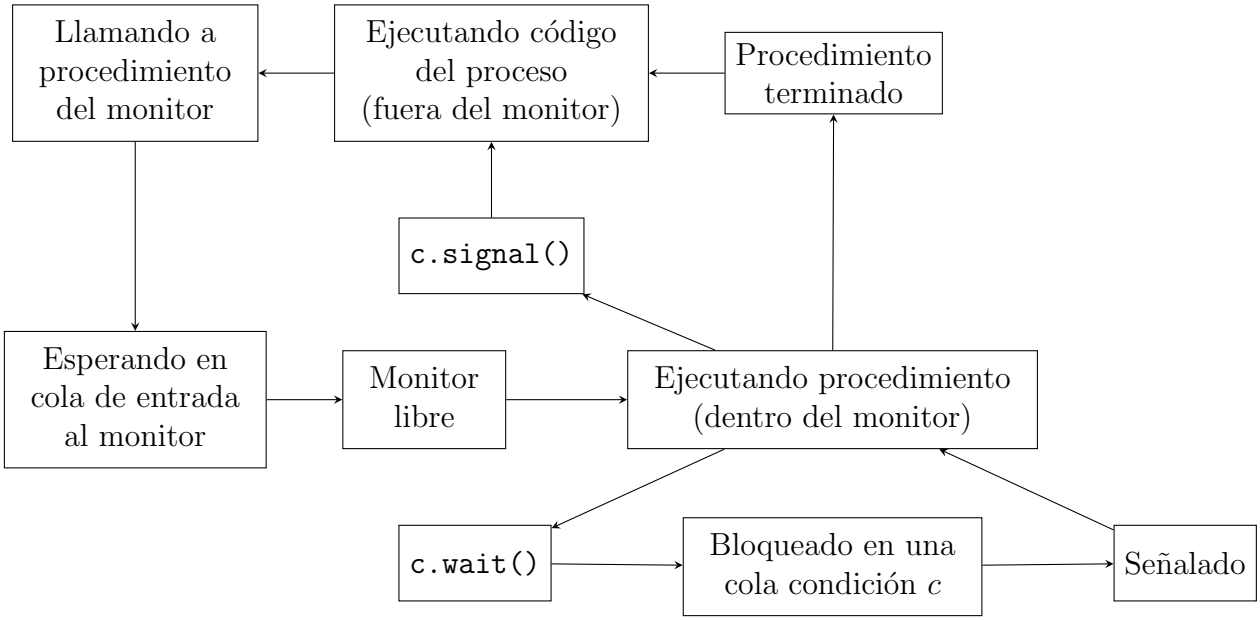


Figura 2.11: Vida de un proceso en un programa con monitores con semántica SS.

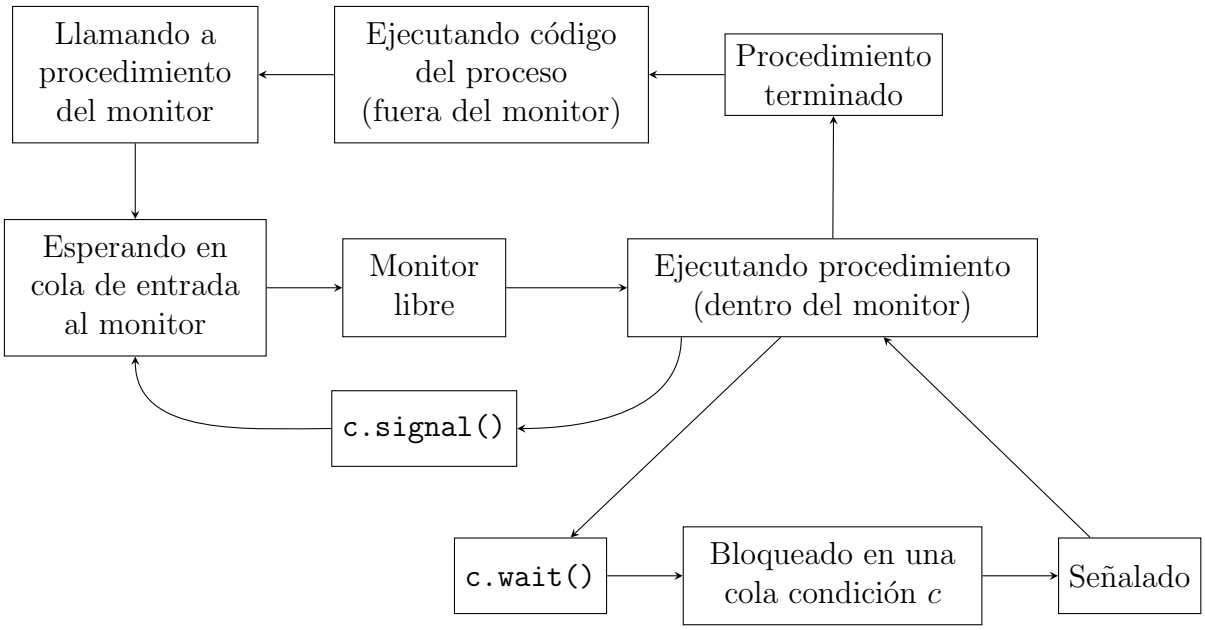


Figura 2.12: Vida de un proceso en un programa con monitores con semántica SE.

2.5.4. Señalar y Espera Urgente (SU)

El proceso señalador se bloquea en una nueva cola del monitor llamada *cola de procesos urgentes*, cediendo el monitor al proceso señalado. El monitor no queda libre en ningún momento, por lo que se trata de una señal con semántica desplazante.

La nueva cola de procesos urgentes actúa como una nueva cola de entrada al monitor, pero con mayor preferencia que la cola de entrada que ya teníamos. De esta forma, es similar a la semántica de señalar y esperar pero sin ser tan *injusta*, ya que da preferencia a los procesos que se bloquearon tras ejecutar un `signal` y deja en segunda posición a los proceos que esperan para ejecutar procedimientos del monitor.

Esta semántica modifica la última representación gráfica que teníamos de un monitor en la Tabla 2.4, dejándola finalmente en la que observamos en la Tabla 2.5.

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

Tabla 2.5: Esquema de un monitor incluyendo la cola de urgentes.

Podemos representar la vida de un proceso que forma parte de un programa concurrente que sua monitores con semántica de señalar y espera urgente con el diagrama de la Figura 2.13.

Finalmente, cabe destacar que con semántica SU, la señal `signal` **siempre** bloquea al proceso señalador en la cola de urgentes, incluso si la cola de dicha variable condición está vacía.

Esto nos permite desbloquear a los procesos de una variable condición de forma que el primer proceso desbloqueado desbloquee al segundo, el segundo al tercero, ..., y así hasta el último, de forma que el último pasará también como el resto a la cola de urgentes, con planificación FIFO, donde el primero volverá a entrar al monitor y así con los siguientes. Notemos que si `signal` no bloquease a los procesos siempre (es decir, si la cola de la variable condición está vacía, no bloqueamos), entonces en el caso anterior la prioridad FIFO de entrada al monitor se invertiría, de forma que el último proceso bloqueado sea el primero que accede al monitor.

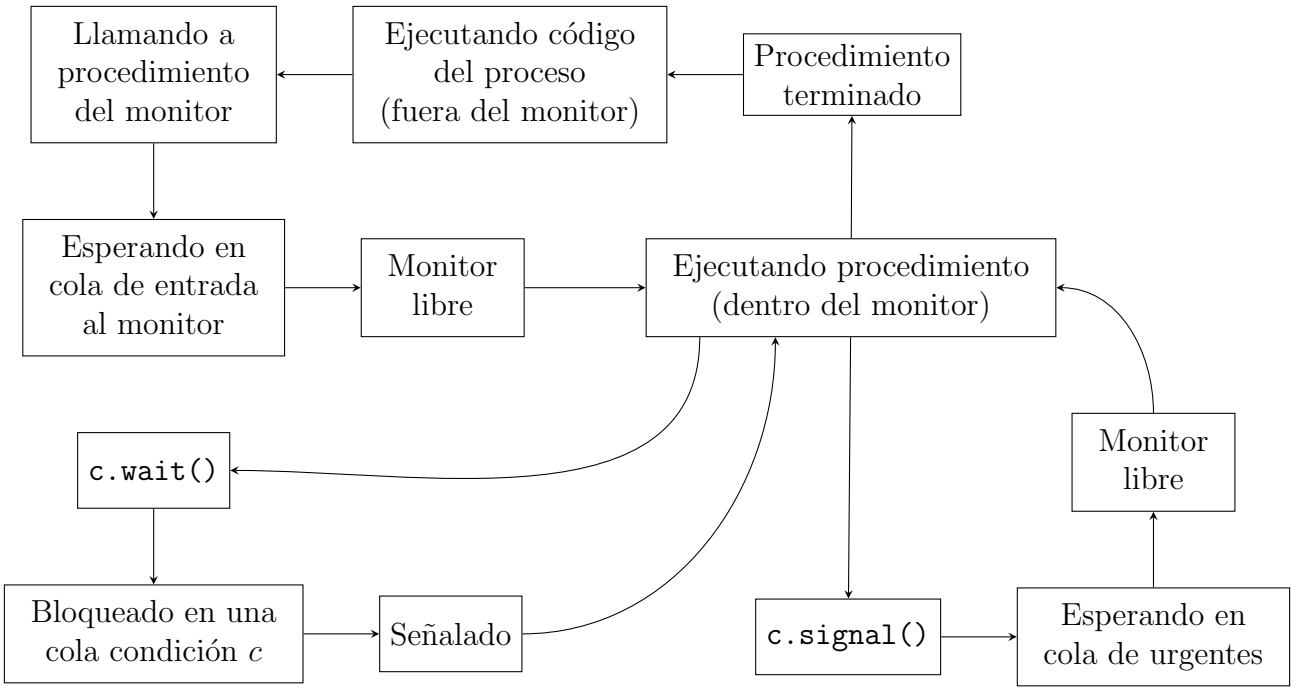


Figura 2.13: Vida de un proceso en un programa con monitores con semántica SU.

2.5.5. Comparativa

Antes de comparar las semánticas de señales que ya hemos descrito, destacamos un último tipo de semántica de señales, las *señales automáticas* (SA). En estas, el programador programa las operaciones `wait` y es el compilador quien analiza los códigos programados y decide cuándo desbloquear a los procesos, por lo que se trata de unas señales implícitas (ya que en ningún moemnto se especifica cuándo hacer una instrucción `signal`).

A pesar de que este tipo de semántica aparece en la bibliografía, no aparece en ningún lenguaje de programación conocido, por lo que no tendrá relevancia en la asignatura ni en la comparativa que ahora realizaremos.

En primer lugar, hemos de destacar que cualquiera de las primeras 4 semánticas vistas es capaz de resolver cualquier problema, por lo que nos guiaremos por la sencillez de uso de cada una y la eficiencia que cada una nos aporta a los programas concurrentes para elegir una u otra semántica.

- En cuato a sencillez de uso, SC, SE y SU presentan la misma facilidad, mientras que SS nos condiciona a un paradigma de programación en el que las instrucciones `signal` sean las últimas que se ejecuten en los procedimientos de un monitor que contenga alguna operación `signal`.

- En cuanto a eficiencia:

  - Las semánticas SE y SU resultan ineficientes cuando las instrucciones `signal` se encuentran al final de los procedimientos (ya que habrá procesos que ejecutan un `signal`, se bloquearán, esperarán en la cola correspondiente de entrada al monitor, y cuando por fin tengan acceso a

él no realizarán nada, sino que simplemente terminarán la ejecución del procedimiento), por lo que se saturará la cola de entrada al monitor y se realizarán cambios de contexto de formas innecesarias.

- La semántica SC es poco eficiente, al tener que obligarnos a usar un bucle de comprobación para comprobar si estamos cerca de una situación insegura cada vez que entramos al monitor, como consecuencia de no tener semántica desplazante, pudiendo producirse robos de señal.

Por tanto, si no nos resulta incómodo tener que colocar todas las instrucciones `signal` como última instrucción de los procedimientos en los que aparecen, SS será probablemente la semántica se señales más eficientes.

Para concluir la comparativa de las semánticas de señales, veremos el siguiente ejemplo, que muestra que a veces la semántica elegida condiciona la forma en la que programamos los monitores.

Ejemplo. Queremos programar en un programa concurrente una *barrera parcial*.

A partir del concepto de barrera que ya manejamos en la asignatura de Arquitectura de Computadores[22], ahora no queremos tener a los procesos esperando a que todos los procesos que intervienen en el programa lleguen a un punto en específico, sino solo queremos que lo hagan n procesos.

De esta forma, cuando el primer proceso llegue al punto de la barrera, este se bloqueará. Estos les sucederá a los primeros $n-1$ procesos en llegar a la barrera. Cuando el proceso n-ésimo llegue a esta, se abrirá la barrera, dejando pasar a estos n primeros procesos que llegaron a esta. La barrera no debe dejar pasar al proceso número $n+1$, sino que este deberá esperar al proceso número $2n$ para voler a abrir la barrera.

Una vez descrito el problema a resolver, planteamos el siguiente monitor como primera solución:

```
1  Monitor BP;
       var cola : cond;
           contador : integer;

5      procedure barrera(); begin
           contador = contador + 1;
           if (contador < n) then
               cola.wait();
           else begin
10             for i=1 to n-1 do
                   cola.signal();
               end
               contador = 0;
           end
15         {Funcionalidad tras pasar la barrera parcial}
       end

       begin
```

[22] Consultar los apuntes si no se conoce el concepto.

```
        contador = 0;
20   end
```

Veamos ahora el comportamiento del monitor, según la semántica de señales que hayamos escogido:

Señalar y Continuar. Cuando llega el n-ésimo proceso de un grupo, este desbloquea a los últimos $n-1$ procesos bloqueados, que pasan a competir por el monitor. Puede suceder que algún proceso del siguiente grupo adelante a uno de este grupo, por lo que no sería una solución válida.

Señalar y Salir. El n-ésimo proceso solo podría despertar al primer proceso que llegó a la barrera (ya que tras ejecutar `signal`, este terminaría la ejecución de su procedimiento), sin reestablecer el contador a 0, por lo que cualquier proceso que pase por la barrera despertaría al siguiente proceso bloqueado, de forma que los últimos n procesos que ejecuten el protocolo `barrera` quedarán bloqueados de forma indefinida.

Señalar y Esperar. El n-ésimo proceso solo podría despertar al primer proceso que llegó a la barrera, volviendo el proceso señalador a la cola de entrada al monitor. El $n+1$-ésimo proceso solo podría despertar al segundo proceso, y se iría a la cola de entrada al monitor. Cuando el $2n$-ésimo proceso entre al monitor, este ya no ejecutará `signal` (ya que como la cola está vacía, es equivalente a una instrucción nula), poniendo por fin el contador a 0 y comenzando otra vez con dicho comportamiento.

No es una solución válida.

Señalar y Espera Urgente. El n-ésimo proceso desbloquea al primer proceso que llegó a la barrera y a continuación se bloquearía en la cola de urgentes, cediendo el monitor al primer proceso, que finalizaría la ejecución del procedimiento y volvería a entrar el n-ésimo proceso, por tener la cola de urgentes mayor prioridad que la cola de entrada al monitor. El proceso se repetiría hasta que el n-ésimo proceso finalie la ejecución del procedimiento, restaurando la barrera parcial para los siguientes n procesos.

Sería un funcionamiento correcto de la barrera, pero hace que el n-ésimo proceso realice $n-1$ cambios de contexto innecesarios.

Mostramos ahora una versión alternativa para el monitor que resuelve el problema de la barrera parcial:

```
1  Monitor BP;
      var cola : cond;
          contador : integer;

5     procedure barrera(); begin
         contador = contador + 1;
         if (contador < n) then
            cola.wait();
         contador = contador - 1;
10       {Funcionalidad tras pasar la barrera parcial}
```

```
      if (contador > 0) then
         cola.signal();
   end

   begin
      contador = 0;
   end
```

Vemos su comportamiento:

- No funciona con la semántica SC.
- Funciona con cualquier semántica desplazante, ya que tras llegar a n procesos en la barrera, el n-ésimo desbloquea al primero, que desbloquea al segundo, que desbloquea al tercero, ..., hasta llegar al $n-1$, que realiza su funcionalidad y viceversa hasta el n-ésimo proceso, que deja la barrera en el estado inicial para los siguientes grupos de procesos.

En general, tenemos que tener cuidado con la semántica de señales que estemos empleando, sobre todo si las instrucciones `signal` tienen instrucciones detrás. Además, la semántica SC suele complicar como norma general los diseños de los monitores.

Ejercicio. Determinar para qué tipo de semánticas (SC o desplazantes) funcionan los siguientes monitores que tratan de simular el comportamiento de un semáforo:

```
Monitor semaforo_FIFO1;
var c : cond;
    s : int;

procedure P;
begin
    if (s = 0) then
        c.wait();
    s := s - 1;
end;

procedure V;
begin
    s := s + 1;
    c.signal();
end

begin
    s := 0;
end
```

```
Monitor semaforo_FIFO2;
var c : cond;
    s : int;

procedure P;
begin
    while (s = 0) do
        c.wait();
    end do;
    s := s - 1;
end;

procedure V;
begin
    c.signal();
    s := s + 1;
end

begin
    s := 0;
end
```

En primer lugar, notamos que el monitor de la izquierda realiza una instrucción `if` antes de llamar a la operación `wait`, luego solo sirve para semánticas desplazantes.

Posteriormente, observamo que el monitor de la derecha realiza la operación `wait` dentro de un bucle, por lo que este sirve para semánticas SC. Sin embargo, no sirve para semánticas desplazantes, ya que la operación `signal` se realiza antes de incrementar la variable `s`, por lo que nunca se realizaría dicho incremento, al bloquear al proceso señalador.

2.5.6. Axiomas para operaciones de sincronización no desplazantes

Como comentamos ya en la Sección 2.3.2, los axiomas que sabemos para demostrar las operaciones de sincronización en monitores solo aplican cuando trabajamos con señales con semánticas desplazantes. Por tanto, si nos encontramos trabajando con semánticas de señales no desplazantes como SC, necesitamos unas nuevas reglas a aplicar.

Axioma de la operación `wait`.

Sea c una variable de tipo condición, L un invariante local del procedimiento en el que nos encontremos y IM el invariante del monitor que usamos para demostrar la corrección del mismo, se verifica que

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

Es decir, antes y después de la operación `wait` solo podemos asegurar que se cumple el IM (ya que podrían producirse robos de señal), por lo que para asegurarnos de que estamos lejos de un estado inseguro del programa, debemos hacer uso de un bucle `while`, tal y como expusimos anteriormente:

```
1 while (cerca de estado inseguro) do begin
     c.wait();
  end
```

Axioma de la operación `signal`.

Como estamos ante una semántica no desplazante, el proceso señalador seguirá ejecutando el procedimiento que estaba ejecutando cuando ejecutó la operación `signal`, por lo que nada habrá cambiado:

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

Por tanto, la operación `signal` tiene el mismo comportamiento que la instrucción nula en semánticas no desplazantes, ya que podemos poner como pre y poscondición el mismo aserto P y el triple será cierto independientemente del aserto escogido.

2.5.7. Intercambio de señales en programas que usan monitores

En el Capítulo 2.5.5, mencionamos que todas las semánticas de señales eran equivalentes para resolver cualquier problema. Esto se debe a que si se cumplen unas

determinadas condiciones, entonces todas las señales (las automáticas, las desplazantes y las no desplazantes) son equivalentes. Esto significa que si estamos trabajando con una semántica, podemos reemplazar las señales de dicha semántica por cierto código que asemeja el comportamiento de otra semántica de señales.

Las condiciones que deben cumplirse dentro de un monitor para que podamos simular una semántica mediante otra son:

1. Se tiene que exigir como poscondición de la operación `wait` el invariante del monitor y nada más estricto (esto es, no podremos usar directamente la condición de sincronización tras la operación `wait`, sino solamente el invariante del monitor[23]).

2. Con señales continuas, cuando realizamos un `signal` no hay desplazamiento, por lo que para que se puedan asemejar a las señales desplazantes, hemos de obligar a que la llamada a `signal` se haga siempre antes de suspender el proceso y salir del monitor.

   Esto es, que la operación `signal` sea la última instrucción en el procedimiento de un monitor o que tras dicha operación haya una instrucción `wait`.

3. Finalmente, para que sean equivalentes los códigos, no podemos usar la operación `signal_all`.

2.5.8. Señales `wait` con prioridad

Como hemos visto hasta ahora, tras ejecutar una operación `wait` sobre una variable condición `c`, pasamos a la cola de bloqueados de la misma, con planificación FIFO. Sin embargo, en algunos lenguajes de programación aparece la operación `wait` con prioridad sobre las variables de tipo condición, con el fin de solucionar situaciones en las que no queremos que tras una operación `signal` se desbloquee el primero de la cola, sino aquel que tuviera más prioridad y lleve más tiempo en la misma.

En el caso descrito, la operación `wait` aceptará un parámetro entero no negativo llamado *prioridad*, de forma que a menor sea el valor de dicho parámetro, mayor prioridad tendrá en la cola de bloqueados. En este caso, la cola FIFO de la variable condición se sustituye por una cola con prioridad.

Mostramos a continuación un ejemplo para motivar por qué algunos lenguajes implementan esta operación `wait`.

Ejemplo. Queremos programar una alarma para procesos de tal manera que el proceso se bloquee al fijar la alarma y que este se desbloquee al cumplirse la hora previamente fijada. Para ello, implementaremos la alarma haciendo uso de un monitor con dos procedimientos:

- `tick`, cableado a la interrupción de reloj del sistema operativo, con el fin de que el reloj sepa el instante en el que se encuentra en cada momento.

[23] Notemos que esto no inclumple el axioma de la operación `wait`, ya que $C \rightarrow IM$.

Proceso	Instante	Llamada	Cola de variable condición
P1	0	`despiertame(10)`	(P1)
P2	1	`despiertame(3)`	(P2 — P1)
P3	2	`despiertame(5)`	(P2 — P3 — P1)
P4	3	`despiertame(1)`	(P2 — P4 — P3 — P1)

Tabla 2.6: Ejemplo de uso del monitor

- `despiertame`, que servirá para que cada proceso fije el momento en el que quiere ser despertado.

Para simplificar el ejemplo, usaremos como unidad de medida un "tick". Planteamos el siguiente monitor como solución al problema, usando señales `wait` con prioridad:

```
1  monitor despertador;
   var ahora : Long_integer;
       despertar : cond;  {variable condición prioritaria}

5  procedure despiertame(n : integer);
   var alarma : Long_integer;
   begin
       alarma := ahora + n;     {hora para despertar}
       while ahora < alarma do
10         despertar.wait(alarma);
       end do;
       despertar.signal();
   end

15 procedure tick();   {cableada a INT CLK}
   begin
       ahora := ahora + 1;
       despertar.signal();
   end
20
   begin
       ahora := 0;
   end
```

De esta forma, lo que estamos haciendo es ordenar la cola de bloqueados de la variable condición `despertar` poniendo al inicio de la misma el primer proceso a desbloquear.

Por ejemplo, ante el uso del monitor por los procesos que observamos en la Tabla 2.6. Cuando la interrupción de reloj del sistema llame al procedimiento `tick`, se desbloqueará al primer proceso de la cola de la variable condición, P2, que saldrá del bucle `while` y desbloqueará al siguiente proceso de la cola, P4, que hará lo mismo, desbloqueando a P3, y como para P3 `ahora < alarma`, volverá a bloquearse, quedando la cola de la variable condición como (P3 — P1).

Notemos que con el uso de la operación `wait` con prioridad minimizamos el número de procesos a desbloquear en cada nuevo instante, ya que tenemos la cola de bloqueados ordenada de forma que al inicio están los procesos que antes se

desbloquearán, con lo que si encontramos el primer proceso que no se desbloquea, hemos terminado en dicho instante de despertar a los procesos.

Si tratamos de programar este mismo problema sin operaciones `wait` con prioridad, sino con la operación `wait` que venimos usando a lo largo de este Capítulo, el procedimiento `despiertame` quedaría como:

```
1  procedure despiertame(n : integer);
   var alarma : Long_integer;
   begin
       alarma := ahora + n;     {hora para despertar}
5      while ahora < alarma do
           despertar.signal();
           despertar.wait();
       end do;
       despertar.signal();
10 end
```

Lo que sucede es que tenemos que estar continuamente desbloqueando a los procesos para comprobar si deben o no despertarse, al no tener ningún orden en dicha cola.

2.6. Implementación de los monitores

Podemos implementar todas las funcionalidades de un monitor usando semáforos. En esta sección, explicaremos cómo implementar un monitor con semántica de señales SU, que nos garantice el comportamiento de un proceso descrito en la Figura 2.13.

Para conseguir simular el comportamiento de un monitor, hemos de conseguir:

- Tener una cola de entrada al monitor, que implementaremos usando un semáforo llamado `mutex`.
- Tener una cola de procesos urgentes, que implementaremos con un semáforo llamado `next`.
- Contabilizar el número de procesos bloqueados en la cola de urgentes[24], con una variable entera `next_count`.
- Implementación de las variables condición.

  Para ello, por cada variable condición que queramos tener en un monitor, crearemos un semáforo nuevo y una variable entera que controle el número de procesos que bloquea dicho semáforo. Además, crearemos unas nuevas funciones `wait` y `signal` (llamadas `x_wait` y `x_signal`) que simulen el comportamiento de las operaciones `wait` y `signal` de las variables compartidas.

Inicializaremos las variables que nos permiten controlar el monitor de la forma:

[24] Con la finalidad de saber si dicha cola está o no vacía

```
mutex := 1;
next := 0;
next_count := 0;
```

Procedimientos del monitor

Cada vez que nos dispongamos a crear un procedimiento nuevo para el monitor, deberemos ejecutar cierto código antes y después del mismo, con la finalidad de garantizar la exclusión mutua dentro del monitor. Para ello, crearemos dos funciones, `entrada` y `salida`, las cuales deberemos invocar antes y después del cuerpo del procedimiento a programar:

```
procedure P();
begin
    entrada();
    {cuerpo del procedimiento}
    salida();
end
```

De esta forma, el código de entrada al monitor sería:

```
procedure entrada();
begin
    sem_wait(mutex);    {garantizar exclusión mutua}
end
```

Y el de salida:

```
procedure salida();
begin
    if(next_count <> 0)    {hay procesos en cola de urgentes}
        sem_signal(next);
    else
        sem_signal(mutex);   {liberamos el monitor}
    end if
end
```

Variables condición

Por cada variable condición a usar dentro del monitor necesitamos tener un par semáforo, entero. Mostramos ahora cómo podemos implementar las operaciones `wait` y `signal` de las variables condición usando semáforos.

Para ello, crearemos una función por cada operación a simular, la cual recibirá dos parámetros: el semáforo que simula la variable condición (`x_sem`) y la cantidad de procesos que dicho semáforo tiene bloqueados (`x_count`):

```
procedure x_wait(x_sem : semaphore, x_count : integer);
begin
    x_count := x_count + 1;  {un bloqueado más}
```

```
    if(next_count <> 0) then  {hay procesos en cola de urgentes}
        sem_signal(next);
    else
        sem_signal(mutex);  {deja libre el monitor}
    end if

    sem_wait(x_sem);
    x_count := x_count - 1;
end
```

```
procedure x_signal(x_sem : semaphore, x_count : integer);
begin
    if(x_count <> 0) then  {si no hay bloqueado no hace nada}
        next_count := next_count + 1;
        sem_signal(x_sem);
        sem_wait(next);
        next_count := next_count - 1;
    end if
end
```

Y con todas estas funciones tenemos **casi** implementados los monitores. Hay que tener en cuenta que los semáforos no tienen una política de planificación fija, sino que es el sistema operativo quien planifica los procesos desbloqueados por el semáforo.

Es necesario por tanto, usar semáforos con colas FIFO para tener totalmente implementados los monitores.

3. Sistemas basados en paso de mensajes

A continuación, dejaremos de estudiar los sistemas en los que disponemos de diversos procesadores que disponen de una memoria compartida común para centrarnos en aquellos sistemas que no disponen de esta facilidad (a los que llamaremos *sistemas distribuidos*), algo que complicará el desarrollo de programas para estos sistemas, ya que solo podremos sincronizar a los distintos procesadores que intervengan en un programa mediante paso de mensajes, los cuales podrán estar implementados bajo distintas semánticas, que serán estudiadas a lo largo de este Capítulo.

3.1. Introducción a la programación distribuida

Como motivación para justificar la existencia de la programación distribuida, veamos el problema del *cuello de botella de Von Neumann*.

Para Von Neumann, un computador clásico asume que las instrucciones de un programa son enviadas desde una unidad de memoria central hasta la Unidad de Central de Procesamiento (o CPU), a través de un bus de interconexión entre estos dos elementos. Nos encontramos con el problema de que las instrucciones que realizan operaciones sobre datos en memoria utilizan este mismo bus para modificar o consultar datos de la misma, con lo que usamos el mismo bus para dos fines distintos: leer instrucciones a ejecutar y ejecutar instrucciones relativas a operaciones en memoria.

Resulta que si queremos una mayor velocidad en la ejecución de las instrucciones por parte de una CPU, este sobre uso del bus limita el rendimiento de la computación, con lo que resulta en un cuello de botella en la aceleración de los sistemas secuenciales.

Si consideramos ahora la programación paralela, ya no tendremos un solo flujo de "trozos de programa" que se envían desde la memoria a la CPU para su ejecución, sino múltiples flujos de los mismos de forma que cada flujo acabe en un procesador distinto para su ejecución al mismo tiempo.

La programación paralela no soluciona el cuello de botella de Von Neumann, pero sí proporciona una aceleración a los programas con respecto a su implementación secuencial análoga, consiguiendo una aceleración ideal de tiempo n si la versión paralela la ejecutamos en n procesadores.

Además, la programación paralela presenta varias ventajas frente a la programación secuencial:

- Los programas paralelos necesitan menos tiempo de ejecución.
- Es más barato el hardware necesario para un programa paralelo en el caso que queramos que tanto un programa paralelo como uno secuencial se ejecuten a la misma velocidad.
- La programación parelala hace menos grave la limitación de velocidad debida al bus entre la memoria y la CPU.

Sin embargo, no todo en programación paralela son ventajas, sino que también debemos aprender a programar según un nuevo paradigma, el cual introduce grandes dificultades a la hora de realizar la depuración de los programas.

3.1.1. Multiprocesamiento

Antes de seguir con la lectura de esta sección, recomendamos la lectura (si no se ha hecho ya) del capítulo de "Arquitecturas Paralelas" de los apuntes de Arquitectura de Computadores.

El *multiprocesamiento* consiste en la utilización de un sistema con dos o más unidades de procesamiento (a las que llamaremos *procesadores*) para ejecutar los programas de una misma aplicación. Desde el punto de vista del sistema, debemos tener la capacidad de gestionar más de un procesador al mismo tiempo, de forma que reasignemos las tareas entre los procesadores durante la ejecución de los programas; así como poder sincronizar los procesadores para realizar determinadas tareas.

Desde el punto de vista del modelo de ejecución de instrucciones de un programa, los procesadores pueden utilizarse para ejecutar una o varias secuencias de instrucciones sobre uno o varios flujos de datos, con lo que es natural clasificar a los sistemas de multiprocesamiento según estas capacidades:

	Instrucción única	Múltiples instrucciones
Datos únicos	SISD	MISD
Múltiples datos	SIMD	MIMD

Tabla 3.1: Taxonomía de Flynn.

De esta forma:

- El modelo SISD es el modelo de computador que sigue la programación secuencial.
- El modelo SIMD permite que los procesadores se sincronicen para realizar la misma instrucción en contextos distintos de memoria, lo que nos permite, por ejemplo, el procesamiento paralelo de vectores de datos[1].

[1] Algo ya visto en Arquitectura de Computadores.

- El modelo MISD permite realizar múltiples instrucciones sobre un mismo conjunto de datos. Posee pocas ventajas y resulta caro de implementar.
- Finalmente, en MIMD podemos dividir un mismo programa en múltiples hilos de ejecución, cada uno de los cuales con su propio estado de procesador y memoria a usar.

Este último tipo de sistema multiprocesador presenta dos grandes familias de computadores cuyas diferencias pasaremos a comentar:

Multiprocesadores

Un multiprocesador es un computador con muchos procesadores individuales, con la ventaja de que hay una parte de la memoria principal que es común a todos los procesadores.

Este tipo de computadores cuenta con un problema principal, que es la falta de escalabilidad, ya que si en algún momento queremos añadir más procesadores al sistema con el objetivo de solucionar problemas de mayor tamaño, no podremos introducir más memoria principal compartida.

Además, el hardware necesario para que los diferentes procesadores accedean a una zona de memoria común es muy caro, ya que hace falta una interconexión muy rápida entre esta y todos los multiprocadores, si no queremos perder la ganancia que hemos ganado al pasar de un sistema uniprocesador a este.

Multicomputadores

Un multicomputador es un sistema con múltiples procesadores de forma que cada uno de ellos tiene una memoria principal independiente, con lo que no existirá una memoria común a todos ellos, sino un sistema de interconexión entre todos los procesadores, que nos permita el paso de mensajes entre estos.

Soluciona el problema de escalabilidad de los multiprocesadores, ya que si queremos introducir más procesadores al sistema, será necesario con introducir el procesador junto con su memoria principal y conectarlo al sistema de interconexión.

Sin embargo, ya no disponemos de herramientas para la sincronización entre los distintos procesadores (que es de lo que trataba el Capítulo anterior), sino que tendremos que idear nuevas estrategias de sincronización basadas en paso de mensajes, lo que hace a la programación de multicomputadores más difícil que la de multiprocesadores, cuya sincronización se resuelve con el uso de monitores.

De esta forma, como principales inconvenientes a la programación paralela y distribuida, tenemos que:

- Necesitamos aprender un nuevo paradigma de programación que no esté basado en memoria compartida, sino en paso de mensajes.
- La depuración de estos programas se vuelve en una tarea casi imposible, al no disponer de una herramienta que nos pueda mostrar en cada instante la

memoria de cada procesador, el estado del procesador y las operaciones que se suceden en el sistema de interconexión.

3.1.2. Multiprogramación SPMD

El modelo que usaremos[2] para programar multicomputadores es el conocido SPMD (*Single Program Multiple Data*), que se encuentra como una solución intermedia entre SIMD y MIMD, de forma que el código que ejecutan los procesos en cada uno de los procesadores es el mismo (es el mismo programa), pero este se ejecuta sobre conjuntos distintos de datos, es decir, cada programa será ejecutado sobre la memoria de cada procesador.

Como características a destacar de SPMD:

- Es una variante del modelo general de MIMD de Flynn.
- No se necesita una arquitectura especial del computador (como sí sucede en SIMD, con la necesidad de disponer de instrucciones vectoriales).
- Los procesadores ejecutan el mismo programa pero de forma independiente.

Tras la compilación y antes de la ejecución de los programas SPMD, dispondremos de un índice que asociaremos a cada uno de los procesadores, con el fin de identificarlos unívocamente. De esta forma, no todos ejecutarán las mismas instrucciones, pero sí el mismo programa, ya que podremos tener instrucciones condicionadas a dicho índice, lo que nos permitirá especializar trozos de código para un procesador en concreto.

Ejemplo. Como un primer ejemplo muy primitivo de programación en SPMD, podemos pensar que trabajamos con 3 procesadores, de forma que a cada uno asignemos una de las siguientes identidades: cliente, trabajador 1 o trabajador 2.

- El valor de las variables `a`, `b` y `c` definidas en el cliente puede ser leído por un trabajador, pero no modificado.
- Las variables `d`, `e` y `f` de los trabajadores pueden ser leídas y cambiadas por el cliente.

El cliente puede acceder a las variables de los trabajadores, distinguiendo entre el trabajador 1 y el 2 con un índice entre llaves. La función `rank` devuelve para cada procesador su índice asociado. Todos los procesos se sincronizan en la marca `spmd`, de forma que el código que se encuentra dentro de este tipo de bloque es ejecutado en todos los nodos (procesadores).

[2]También podemos encontrar el modelo MPMD, donde por cada procesador que intervenga en el problema a resolver deberemos crear un programa distinto que se ejecutará sobre dicho procesador. Podemos encontrar además modelos mixtos entre SPMD y MPMD.

Código	Cliente			Trab. 1			Trab. 2		
	a	b	c	d	e	f	d	e	f
`a = 3;`	3	-	-	-	-	-	-	-	-
`b = 4;`	3	4	-	-	-	-	-	-	-
`spmd`									
`d = rank();`	3	4	-	1	-	-	2	-	-
`e = d + a;`	3	4	-	1	4	-	2	5	-
`end`									
`c = a + e{1};`	3	4	7	1	4	-	2	5	-
`d{2} = 5;`	3	4	7	1	4	-	5	5	-
`spmd`									
`f = d * b;`	3	4	7	1	4	4	5	5	20
`end`									

Tabla 3.2: Ejemplo de programa SPMD.

3.2. Semántica de las operaciones de paso de mensajes

Como hemos comentado anteriormente, en los multicomputadores no dispondremos de una memoria común a todos los procesadores, pero sí de un sistema de interconexión de todos los procesadores que nos permita comunicarlos mediante mensajes con el fin de sincronizarlos.

Tendremos, por tanto, dos operaciones de comunicación principales para trabajar con los mensajes: las funciones `send` y `receive`.

El significado (o semántica) de dichas operaciones puede ser diferentes, es decir, el resultado de su ejecución puede variar dependiendo de su implementación. Sin embargo, podemos clasificar el tipo de operaciones `send` y `receive` que nos podemos encontrar en relación a si estas cumplen o no la propiedad de seguridad y según el modo de comunicación de las mismas:

Propiedad de seguridad.

Decimos que la propiedad de seguridad en el paso de mensajes se cumple en un programa con la operación `send` cuando la ejecución de esta garantiza que el valor recibido por el destinatario sea el valor que tenían los datos antes de la llamada.

Sin embargo, una operación `send` que no cumpla la propiedad de seguridad (sea insegura) podría ocasionar que el valor recibido por el otro proceso no coincida con el valor de los datos antes de la llamada. Por ejemplo, como resultado de modificar los datos tras realizar dicha llamada pero antes de que el sistema comience a transmitir el valor de dichos datos.

Además, decimos que la propiedad de seguridad en el paso de mensajes se cumple en un programa con la operación `receive` si tras la ejecución de esta disponemos instantáneamente de los datos a recibir en el mensaje asociado.

Puede suceder que haya implementaciones de la operación `receive` que no bloqueen al proceso hasta recibir los datos, con lo que hasta que el mensaje no sea recibido, no podremos usar los datos que esperamos recibir. Esta se trata de una operación `receive` insegura.

Modo de comunicación de las operaciones con paso de mensajes.
Podemos encontrarnos con:

- Operaciones bloqueantes (o síncronas).
- Operaciones no bloqueantes (o asíncronas).

3.2.1. Operaciones bloqueantes

La semántica de este tipo de operaciones de paso de mensajes quiere decir que la operación `send` solo volverá (terminará su ejecución) cuando se garantice la propiedad de seguridad anteriormente vista. Es decir, la operación `send` terminará cuando se haya transmitido en mensaje, con lo que la alteración de alguno de los valores que intervienen en el mensaje no modificará a su vez su valor dentro del mensaje. Cuando la operación `send` termine, no podemos asegurar que el receptor haya recibido el mensaje, sino solo que este se ha mandado de forma segura.

Encontramos diferentes tipos de operaciones bloqueantes, en relación a su modo de comunicación (si hay un buffer intermedio o si no) y a si hay un hardware especializado o si no:

Modo de comunicación	Hardware especializado	Sincronización	Seguridad
Sin buffer	-	Sí (con "citas")	Sí
Con buffer	Sí	Relajada	Sí
	No	Sí	Sí

Tabla 3.3: Tipos de operaciones bloqueantes.

Paso de mensajes síncrono (sin buffer)

Si no disponemos de un buffer intermedio para la comunicación, esta se lleva a cabo mediante un enlace directo entre los dos procesos que participan. Antes de que los datos se transmitan de forma física, ambos procesos han de estar preparados para realizar el intercambio, lo cual exige una *cita* entre el emisor y el receptor. Es decir:

- Si el emisor ejecuta la función `send` antes de de que el receptor haya ejecutado `receive`, el emisor ha de bloquearse hasta que termine la transmisión de los datos (es decir, hasta que el receptor complete la ejecución de `receive`).
- Si el receptor ejecuta la función `receive` antes de que el emisor haya ejecutado `send`, el receptor ha de bloquearse hasta que el emisor haya ejecutado `send` y disponga de los datos.

Por tanto, el mecanismo de *citas* implica:

- Sincronización entre el emisor y el receptor para que se produzca el intercambio.
- La posibilidad de que el emisor realice asertos acerca del estado del receptor en el punto de sincronización, lo que permitiría extender el sistema de verificación la *Lógica de Programas* a los sistemas distribuidos con este modo de comunicación.

Aunque este sistema proporciona a los programas un tipo de comunicación que respeta la semántica de seguridad, suele ser una implementación ineficiente, ya que cada vez que se ejecute un `send` o `receive`, probablemente se tendrá que esperar una cantidad de tiempo no despreciable.

Además, este modelo puede llevar a una situación de interbloqueo entre dos o más procesos:

```
Process P0;
var x0;
begin
   send(&dato1, P1);
   receive(&x0, P1);
end
```

```
Process P1;
var x1;
begin
   send(&dato2, P0);
   receive(&x1, P0);
end
```

Figura 3.1: Situación de interbloqueo en el mecanismo de citas.

Donde `P0` debe esperar a que `P1` realice el `receive` asociado a `dato1`, mientras que `P1` debe esperar a que `P0` realice el `receive` asociado a `dato2`.

Paso de mensajes con buffer

En este tipo de paso de mensajes, la operación `receive` posee la misma semántica que en el caso anterior, ya que no es lógico que esta termine cuando todavía el otro proceso no ha ejecutado el `send` correspondiente, con lo que hay que esperar a que ejecute dicha función y que además se reciba el mensaje completamente.

Lo que cambia, por tanto, en este tipo de mensajes es la operación `send`. Como el medio de comunicación entre los procesos no es ahora un enlace directo si no una cola de mensajes gracias al buffer intermedio, esta operación `send` no debe ya esperar a la ejecución de la instrucción `receive` en el receptor, sino solo de la copia del mensaje enviado en el buffer intermedio.
En el caso en el que dicho buffer se encuentre lleno, se deberá esperar a que se libere un mensaje, con lo que en dicho caso, la situación es totalmente análoga al mecanismo de citas.

De esta forma, al ejecutar la operación `receive` lo que se hace es consultar el buffer intermedio y:

- Si hay un mensaje en dicho buffer, se obtiene el mensaje, con lo que obtenemos los datos transmitidos.
- Si no hay un mensaje en dicho buffer, se deberá esperar a la correspondiente operación `send` que introduzca el mensaje en el buffer.

Contamos con dos variantes a mencionar en el modo de comunicación con buffer intermedio:

Sin hardware especializado.
La situación es la descrita superiormente, aunque contamos con una operación `vacio` que nos dice si el buffer intermedio se encuentra vacío o no, con lo que en lugar de hacer en el receptor una espera ociosa hasta que haya algún mensaje, podemos realizar trabajo útil mientras no se necesite la información de dicho buffer.

Con hardware especializado.
Hay una variante de lo descrito anteriormente y es el caso de disponer de un buffer interno de longitud fija en el receptor. En este tipo de sistemas, la operación `send` bloquea solo cuando se intenta añadir un mensaje a un buffer intermedio que ya está lleno.

Por otra parte, disponemos de hardware específico en el receptor que copia el contenido del buffer intermedio al buffer interno, de forma que cuando este ejecute la operación `receive`, solo se tendrá que copiar la información del buffer interno a la zona de memoria asignada a la recepción de mensajes.

Puede contarse además con hardware especializado que tras la copia de los mensajes en el buffer intermedio, copie inmediatamente dichos mensajes a la zona de memoria asignada al proceso, con lo que diremos que la sincronización entre los procesos emisor y receptor se verá *relajada*.

Si no se cuenta con dicho hardware especializado, el proceso receptor deberá interrumpirse al llamar a `receive`, para intervenir en la transferencia interna de datos a su memoria.

En cualquier caso, el uso de un buffer intermedio entre los dos procesos resulta en una comunicación más rápida que con el mecanismo de citas.

Sin embargo, la situación de interbloqueo anteriormente comentada también se puede llegar a dar.

3.2.2. Operaciones no bloqueantes

Las operaciones bloqueantes garantizan comunicaciones con semántica segura respecto a los datos que transmiten, pero dicha seguridad la pagamos con la ineficiencia a la hora de su implementación en las plataformas que no poseen de un hardware de comunicaciones especializado, ya que:

- El mecanismo de citas requiere de la espera ociosa por parte de un proceso en la comunicación.
- El mecanismo de buffer intermedio introduce una sobrecarga debida a la gestión del propio buffer y de una posible sincronización interna.

Como solución para evitar la ineficiencia introducida por las operaciones bloqueantes, consideramos dejar como responsabilidad para el programador asegurar que los datos no sean alterados mientras que estos están siendo transmitidos, con el objetivo de ganar velocidad de ejecución en los programas a realizar.

De esta forma, las operaciones `send` y `receive` terminarán casi inmediatamente, antes de que sea seguro modificar (en el emisor) o usar (en el receptor) los datos, de modo que el programador sea el responsable de asegurar que no sean modificados mientras estos están siendo transmitidos entre los procesos.

Para poder llevar esto a cabo, es necesaria la existencia de sentencias de comprobación del estado del envío de los mensajes, que indiquen si en un momento dado se pueden alterar los datos sin provocar que la operación de paso de mensajes deje de ser segura.

De esta forma, una vez iniciada la comunicación entre los procesos, el programa podría realizar cualquier otro cálculo que no necesite del uso de los datos que intervengan en el mensaje en cuestión, comprobando la terminación de la operación de comunicación cuando sea necesario.

Paso de mensajes sin buffer

En resumen, la operación `send` volverá inmediatamente, de forma que solo indique al sistema en cuestión que ha de realizar una transmisión de un determinado mensaje a otro procesador. Tras la ejecución de `send` y mientras que el mensaje no sea transmitido, habrá un periodo de "inseguridad" durante el cual los datos del mensaje no deben ser modificados, si queremos que se cumpla la propiedad de seguridad de los mismos.

Por otra parte, la vuelta inmediata de la operación `receive` dependerá de si contamos o no con hardware especializado:

Si existe hardware especializado.
: El proceso receptor no se bloqueará al llamar a la operación `receive`, aunque no se hayan terminado de transmitir los datos al receptor, con lo que tendremos un periodo de tiempo en el que acceder a ciertos datos serán unas instrucciones inseguras, con lo que debemos contar con operaciones de comprobación que nos indiquen cuándo es seguro acceder a los datos que están siendo transmitidos.

Si no existe hardware especializado.
: El proceso receptor se ha de suspender al llamar a la operación `receive`, desde que el sistema esté preparado para recibir los datos hasta el final de la transmisión de los mismos, para que se pueda garantizar la semántica segura de las operaciones de paso de mensajes.

Paso de mensajes con buffer

Ahora, cuando se llame a la operación `receive`, no habrá interrupcción del proceso, pero se comenzará con la transferencia de los datos del mensaje a quien aplica

esta operación desde un buffer de recepción interno del sistema al área de memoria donde se espera recibirlos.

Como consecuecnia, se reduce el tiempo de espera en el proceso receptor, aunque el acceso a los datos durante la copia de los mismos es insegura.

3.3. Diseño de programas distribuidos

A continuación, veremos los roles predominantes que tienen los procesos en los sistemas distribuidos, así como una nueva sentencia de programación necesaria para estos sistemas.

3.3.1. Tipos de procesos

En un sistema distribuido, es normal asociar "roles" a los procesos que intervienen en la ejecución del sistema, con la finalidad de explicar su existencia o compartamiento. Distinguimos 4 roles principales:

Filtros. Son procesos transformadores de datos: Operan sobre un flujo de datos entrante de forma que para cada dato de entrada (o conjunto de datos de entrada) es capaz de emitir un dato en el flujo de datos de salida.

En programas destinados a la realización de cálculos, es normal contar con varios procesos filtros, de forma que un proceso filtro alimente a otro filtro con su salida.

Clientes. Son procesos desencadenantes de algo: Hacen peticiones a otros procesos, desencadenando en estos un comportamiento esperado. Muchas veces esperan a ser respondidos.

Servidores. Son procesos reactivos: esperan a que se les hagan peticiones y reaccionan a estas, desencadenando las acciones adecuadas o respondiendo al proceso que realizó la petición. Suele ser un proceso que nunca termina y que da servicio a muchos procesos.

Es común observar en programación la arquitectura cliente/servidor, de forma que dispongamos de varios procesos clientes que tengan que hacer uso de un servidor para completar su funcionalidad.

Pares. Decimos que dos o varios procesos son pares si son idénticos y colaboran entre sí para resolver un problema. Son comunes en las arquitecturas *peer-to-peer*, donde solo tenemos procesos pares con el fin de resolver el problema en cuestión.

3.3.2. Órdenes con guarda

Como motivación a las instrucciones de órdenes con guarda, planteamos el *problema del museo* en el siguiente ejemplo:

```
Process P1;
begin
while true do begin
   while(no_pasa_persona()) do begin
      null;
   end
   send(aviso, P3);
end
end
```

```
Process P2;
begin
while true do begin
   while(no_pasa_persona()) do begin
      null;
   end
   send(aviso, P3);
end
end
```

Figura 3.2: Código para las puertas del museo.

```
P3;
var aforo : integer;
begin
while true do begin
   receive(aviso, P1);
   aforo := aforo + 1;
   receive(aviso, P2);
   aforo := aforo + 1;

   { Resto de sentencias }
end
end
```

Figura 3.3: Primer código para el controlador.

Ejemplo. En un museo, contamos con dos puertas giratorias, ambas controladas por sensores infrarrojos que tienen la finalidad de avisar a un controlador cuando detectan el paso de personas, para controlar la entrada al museo. De esta forma, el código de los procesos que son responsables de las puertas puede ser similar al de la Figura 3.2.

Tendremos otro proceso P_3 que hará las veces de controlador: debe controlar el aforo del museo, así como no contar aquellas personas que pasen tras la hora de cierre. Para ello, debe ser capaz de recibir los mensajes de las dos puertas (de los procesos P_1 y P_2). Un primer código que se nos podría ocurrir para el controlador es el de la Figura 3.3 Sin embargo, este código no permite detectar a personas por la segunda puerta antes de detectar alguna persona por la primera. Además, no permite que dos personas pasen juntas por la misma puerta, sino que obliga a la alternancia obligatoria entre las puertas, teniendo que pasar primero por la primera puerta, un comportamiento que no queremos obligar a los visitantes del museo.

Ejemplo. De forma similar al ejemplo anterior, podemos tener un proceso servidor que dé servicio a 100 clientes (o 1000, o cualquier número de clientes). Nos preguntamos por un código que nos permita recibir un mensaje de cualquier cliente y luego actuar en base al mensaje recibido.

Sin embargo, al igual que pasaba con el museo, no obtenemos respuesta alguna sobre cómo podemos programar esta funcionalidad, ya que debemos exigir un orden en cómo el proceso receptor de mensajes debe recibirlos.

Definición

Estos dos ejemplos sirven de motivación para la existencia de las órdenes con guarda. Estas son un tipo de órdenes delimitadas por las palabras reservadas `if` (al inicio) y `fi` (al final), y separadas por un delimitador (en nuestro caso, `[]`).

Una orden con guarda consta de una orden (cualquier sentencia del lenguaje de programación) y de una guarda[3], es decir, una condición booleana.

De esta forma, cuando el proceso que está ejecutando el programa se encuentra con una orden con guarda, lo que hará será evaluar todas las guardas de la misma, quedándose con las que se evalúan como verdaderas, y escogerá de forma no determinista una de ellas para ejecutar su orden asociada.

Ejemplo. Recuperando el ejemplo del problema del museo, podemos hacer uso de una órden con guarda para resolver el problema de que no sabíamos cómo programar el proceso P_3 que actúa de controlador del museo. Haciendo uso de las órdenes con guarda, el código del mismo queda como el de la Figura 3.4.

```
P3;
var aforo : integer;
begin
while true do begin
   if
      (receive(aviso, P1))
   []
      (receive(aviso, P2))
   fi

   aforo := aforo + 1;
   { Resto de sentencias }
end
end
```

Figura 3.4: Código para el controlador usando órdenes con guarda.

De esta forma, en el caso de que las operaciones `receive` devolvieran `true` si tienen el mensaje disponible y `false` en caso contrario, tenemos un código para el controlador del museo.

3.4. Espera selectiva

Tras ver en la Sección 3.3.2 las órdenes con guarda, procedemos ahora a definir y explicar el comportamiento de la espera selectiva, un tipo de orden con guarda

[3] Quién lo habría dicho.

que poseen ciertos lenguajes de programación de sistemas distribuidos con señales síncronas, con el fin de solucionar el problema de la recepción de múltiples mensajes.

3.4.1. Definición de sentencia de espera selectiva o `select`

Una sentencia de espera activa comienza con la palabra reservada `select` y termina con la palabra reservada `end`, de forma que entre estas dos palabras cuenta con varias alternativas u órdenes con guarda.

Una orden con guarda comienza con la palabra reservada `when`, y en ella podemos encontrar dos partes bien diferenciadas:

- La guarda, conformada por una condición booleana y una instrucción `receive`. Se sitúan inmediatamente después del `when`.
- Un bloque de código, que se sitúa tras la palabra reservada `do`, situada esta tras la guarda.

En el caso de la espera selectiva, encontramos tres alternativas para la guarda:

- Que sea una condición booleana.
- Que sea una condición booleana y una instrucción `receive`.

  De esta forma, si la condición booleana es evaluada como `false`, la instrucción `receive` no llegará a ejecutarse. Este tipo de guardas son frecuentemente llamadas "guardas sin sentencias de entrada".
- Que sea una instrucción `receive`.

  El comportamiento de este tipo de guarda es idéntico al que obtendríamos con una guarda con la misma operación `receive` y la condición `true`.

Es decir, la sintaxis de una espera selectiva es:

```
select
   when condicion1 receive(variable1, proceso1) do
      sentencias;
   when condicion2 receive(variable2, proceso2) do
      sentencias;
   ...
   when condicionn receive(variablen, proceson) do
      sentencias;
end
```

Puediendo omitir en cada guarda la condición booleana o la instrucción `receive`, pero no ambas.

La razón por la que solo permitimos instrucciones `receive` en las guardas y no instrucciones `send` es debido a que no queremos que la evaluación de las alternativas de la instrucción `select` cambie el estado del proceso, así como evitar situaciones de bloqueos, las cuales son más frecuentes con el uso de instrucciones `send`.

3.4.2. Comportamiento de la espera selectiva

Cuando un proceso se encuentra ejecutando su código y se encuentra con una espera selectiva o sentencia `select`, lo que hace es evaluar primero todas las guardas, con el fin de clasificar a los guardas de las órdenes con guarda en tres tipos (estos tipos son dinámicos, de forma que una orden con guarda puede tener un tipo en una ejecución de la sentencia `select` y que tenga un tipo distinto en la siguiente ejecución de la misma sentencia):

Guardas ejecutables. Una guarda es ejecutable si su condición es evaluada como cierta y además el proceso cuyo nombre se indica en la instrucción `receive` ya terminó de enviar el mensaje (es decir, el mensaje está "listo" para ser recibido).

Si la guarda no tiene instrucción `receive`, bastará con que su condición booleana sea evaluada como cierta[4]. En caso de no disponer de condición booleana, entendemos que tiene la condición `true`.

Guardas potencialmente ejecutables. Decimos que una guarda es potencialmente ejecutable cuando su condición booleana es evaluada como cierta pero todavía el mensaje no ha sido enviado (es decir, si la ejecución de la instrucción `receive` provoque que el proceso receptor se bloquee esperando la correspondiente instrucción `send`).

Si una guarda no tiene instrucción `receive` no puede ser potencialmente ejecutable. En caso de no tener condición booleana, se entiende que su condición es `true`.

Guardas no ejecutables. Una guarda se dice que es no ejecutable si su condición booleana es evaluada como falsa.

De esta forma, las guardas sin condiciones booleanas nunca son no ejecutables.

Una vez clasificadas en tiempo de ejecución todas las guardas, es necesario seleccionar una orden con guarda para su ejecución. El criterio de selección del bloque de código de dicha orden es el siguiente:

1. Si hay una o varias guardas ejecutables con instrucción `receive`, se selecciona la orden con guarda correspondiente a la instrucción `receive` cuyo proceso emisor realizó primero la instrucción `send`.

2. Si hay una o varias guardas ejecutables pero ninguna tiene sentencia de entrada, se selecciona de forma no determinista una de ellas para su ejecución.

3. Si no hay guardas ejecutables pero hay una o varias guardas potencialmente ejecutables, entonces el proceso se **bloqueará** hasta que alguna de las guardas pase a ser ejecutable, en cuyo caso ejecutará dicha orden con guarda.

4. Si todas las guardas son no ejecutables, la instrucción terminará con un error.

Destacamos ciertas consecuencias de este comportamiento de la instrucción:

[4]Podemos pensar que el razonamiento anterior es cierto por vacuidad.

- Debido a que es deseable que no todas las guardas sean no ejecutables, debemos establecer las condiciones de las guardas de forma que la disyunción de todas ellas sea un aserto. Es decir, debemos intentar tener en las condiciones de las guardas todas las alternativas que puedan plantearse en el problema.

- La ejecución de una instrucción `select` puede conllevar esperas, en el caso de que todas las guardas sean potencialmente ejecutables.

- Los bloques de código de cada orden con guarda se ejecutan de principio a fin.

- Hay versiones de esperas selectivas con prioridad, de forma que podamos establecer una prioridad en la selección de la guarda a ejecutar.

Ejemplo. Para ilustrar el uso de la sentencia `select`, resolveremos ahora el problema de los productores/consumidores en un sistema distribuido. Para ello, dispondremos de un proceso productor, que generará datos, y de un proeceso consumidor, que procesará dichos datos. Además, como no contamos con una memoria compartida, será necesaria la creación de un nuevo proceso, que sirva como intermediario entre el productor y el consumidor.

De esta forma, el código del productor y del consumidor puede ser el observado en la Figura 3.5.

```
{Identificador: P}
Process Productor;
var v : integer;
begin
while true do begin
   v := Producir();
   send(v, B);
end
end
```

```
{Identificador: C}
Process Consumidor;
var v : integer;
begin
while true do begin
   send(s, B);
   receive(v, B);
   Consumir(v);
end
end
```

Figura 3.5: Código para el problema de los productores/consumidores.

Así, el productir enviará los valores producidos al proceso intermedio (identificado por `B`), y el consumidor deberá solicitar datos al proceso intermedio, quien responderá con dichos datos.

Es fácil adivinar que en el código de dicho proceso intermedio debe aparecer una instrucción `select`, así como un buffer de dicho proceso para almacenar los datos producidos mientras que no están siendo consumidos. El código del proceso intermedio es el que se observa en la Figura 3.6.

```
{Identificador: B}
Process Intermedio;
var esc, lec, ocupados : integer := 0;
    buf : array[0..tam-1] of integer;
begin
while true do begin
   select
      when cont < tam receive(v, P) do
         buf[esc] := v;
         esc := (esc + 1) mod tam;
         ocupados := ocupados + 1;
      when cont > tam receive(s, C) do
         send(buf[lec], C);
         lec := (lec + 1) mod tam;
         ocupados := ocupados - 1;
   end
end
end
```

Figura 3.6: Proceso intermedio en el problema de los productores/consumidores.

3.4.3. `select` con guardas indexadas

Puede suceder que tengamos un proceso servidor que dé servicio a miles de clientes de forma que la acción a realizar por parte del servidor sea idéntica en muchos de esos clientes (como por ejemplo en el caso del problema de productores/consumidores con múltiples productores y múltiples consumidores).

En dicho caso, no será necesario programar miles de órdenes con guarda, sino sólamente una, que dependerá de un índice, cuyo rango de variación estará acotado. De esta forma, introducimos la sintaxis:

```
for indice := inicial to final
   when condicion(indice) receive(variable(indice), fuente(indice)) do
      sentencias(indice)
```

como una orden con guarda válida, de forma que esta se sustituirá por $final - inicial$ órdenes con guarda, de la forma:

```
when condicion(inicial) receive(variable(inicial), fuente(inicial)) do
   sentencias(inicial)
when condicion(inicial+1) receive(variable(inicial+1), fuente(inicial+1)) do
   sentencias(inicial+1)
...
when condicion(final) receive(variable(final), fuente(final)) do
   sentencias(final)
```

Ejemplo. Si queremos programar un proceso servidor que vaya acumulando la suma que le ordenen n procesos hasta que la suma de cada proceso iguale o supere el valor 100, podemos programar una sentencia `select` que haga uso de las guardas indexadas, tal y como vemos en la Figura 3.7.

```
select
   for i := 0 to n-1
      when suma[i] < 100 receive(numero, fuente[i]) do
         suma[i] := suma[i] + numero;
end
```

Figura 3.7: Código con guardas indexadas.

Este código será equivalente al de la Figura 3.8

```
select
   when suma[0] < 100 receive(numero, fuente[0]) do
      suma[0] := suma[0] + numero;
   when suma[1] < 100 receive(numero, fuente[1]) do
      suma[1] := suma[1] + numero;
   ...
   when suma[n-1] < 100 receive(numero, fuente[n-1]) do
      suma[n-1] := suma[n-1] + numero;
end
```

Figura 3.8: Código equivalente.

3.4.4. `select` con sentencia `else`

Al igual que en la operación `switch` (de `C++` o Java) contamos con la palabra reservada `default` para no dejarnos ningún caso, en las sentencias `select` podemos hacer uso de la palabra reservada `else` para que, en el caso de que todas las guardas sean no ejecutables una vez que el proeso alcanza la sentencia `select`, que esta no termine con error, sino que ejecute el bloque de código asociado a la palabra `else`, de forma que introducimos la sintaxis:

```
else
   sentencias;
```

como una operación con guarda válida más, la cual será ejecutada si el resto de órdenes con guarda son no ejecutables cuando el proceso se disponga a ejecutar la sentencia `select`.

Ejemplo. Recuperando el ejemplo anterior del proceso servidor que reúne la suma de n procesos, mostramos ahora el código completo del proceso servidor, en la Figura 3.9, al disponer ya de la sentencia `else`.

```
1  Process Servidor;
   var suma : array[0..n-1] of integer := (0,0,...,0);
       continuar : boolean := true;
       numero : integer;
5  begin
      while continuar do begin
         select
            for i := 0 to n-1
               when suma[i] < 100 receive(numero, fuente[i]) do
10                suma[i] := suma[i] + numero;
            else
               continuar := false;
         end
      end
15 end
```

Figura 3.9: Código del proceso servidor.

4. Relaciones de problemas

4.1. Introducción

Ejercicio 4.1.1. 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 4.1.2. ¿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:

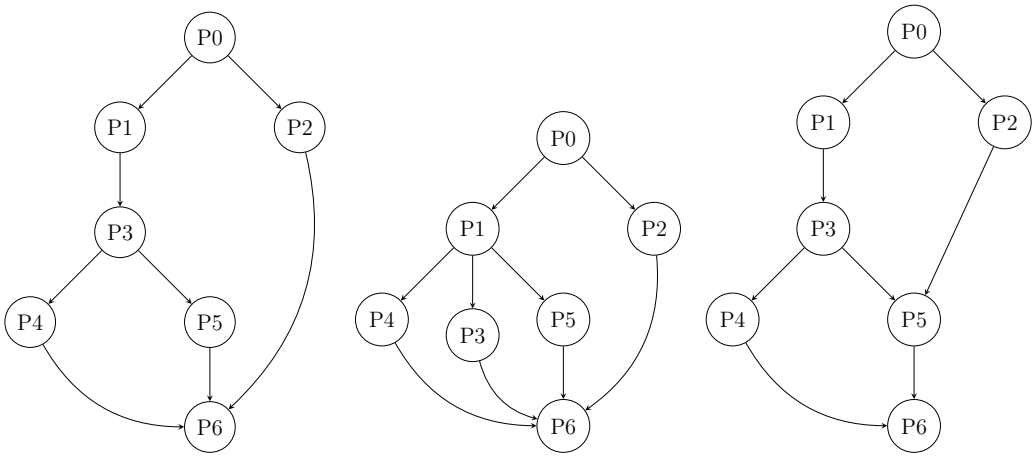
- Uno que lea un ítem del fichero `f` y lo escriba en una variable compartida.
- Otro que lea dicha variable compartida y escriba el ítem en el fichero `g`.

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, ant) ;
      sig = leer(f) ;
    coend
  end
end
```



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

Figura 4.1: Grafos de precedencia del ejercicio 4.1.3.

Ejercicio 4.1.3. 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 4.1.

1. Grafo de precedencia de la figura 4.1a:

```
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 4.1b:

```
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 4.1c:

```
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 4.1c, 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 4.1.4. 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 4.4: Programas concurrentes del ejercicio 4.1.4.

1. Programa de la figura 4.4a.

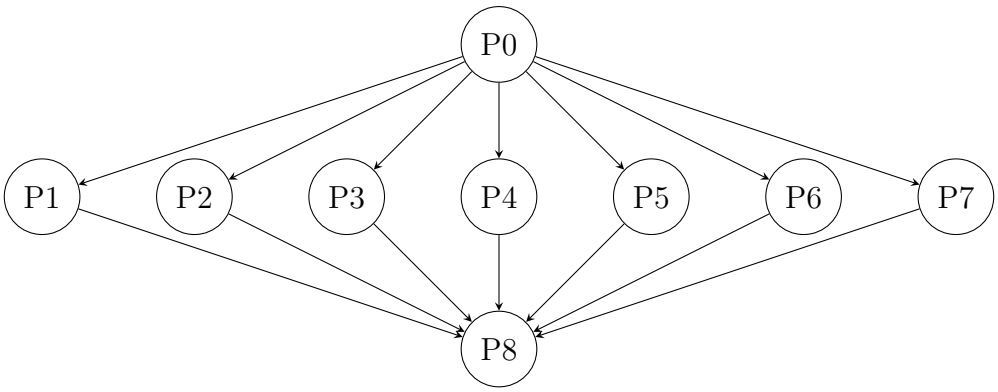


Figura 4.5: DAG para la figura 4.4a.

2. Programa de la figura 4.4b.

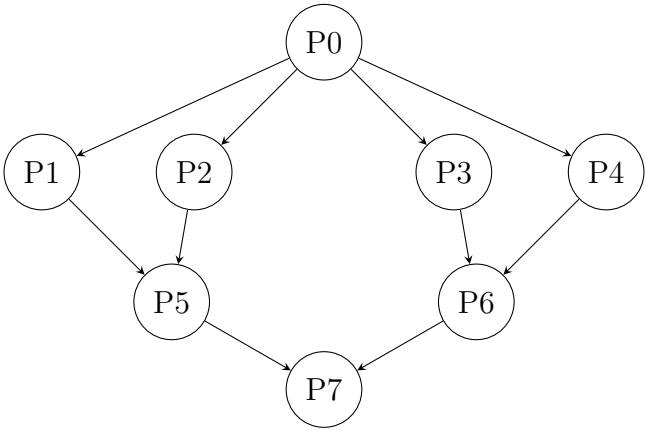


Figura 4.6: DAG para la figura 4.4b.

Ejercicio 4.1.5. 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 4.1. 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 4.1.5.

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 4.1: Tabla de opciones del ejercicio 4.1.5.

```
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 4.1.6.

Ejercicio 4.1.6. 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 4.1.6.

```
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 4.1.6.

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 4.1.7. 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.

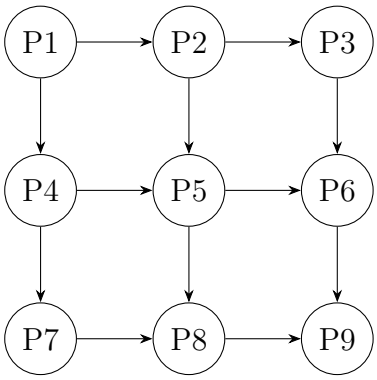
```
var a, b, c : array[1..3,1..3] of
real ;
process CalcularColumna[j : 1..3] ;
    var i, k : integer ;
    begin
        for i := 1 to 3 do begin
            c[i,j] := 0 ;
            for k := 1 to 3 do
                c[i,j] := c[i,j] +
                a[i,k]*b[k,j] ;
        end
    end
end
```

(b) Procesos concurrentes para calcular por columnas.

Figura 4.7: Códigos para el ejercicio 4.1.7.

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

(a) Código del ejercicio 4.1.8.



(b) DAG del ejercicio 4.1.8.

Figura 4.8: Figuras del ejercicio 4.1.8.

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 4.7a) o bien cada uno puede calcular una columna de la matriz `c` (Código Fuente 4.7b). 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 4.1.8. 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 4.8a), pero que requieren sincronizarse según determinado grafo (ver la figura 4.8b).

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:

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

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

```
1 procedure EsperarPor( i : integer ) ;
  begin
      while not finalizado[i] do
      begin
5         { 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 se 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`.

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

Ejercicio 4.1.9. En el ejercicio 4.1.8 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 4.1.10. 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 4.1.11. ¿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 4.1.12. 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 4.1.13. 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 4.1.14. 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 4.1.15. 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 4.1.16. 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 4.1.17. 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 4.1.18. 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 4.1.19. 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 4.1.20. 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 4.1.21. 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:

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

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

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

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 $\{x = a\}$.

Observación. Notemos que esto puede parecer contraintuitivo, ya que el lector sabe que ese triple es cierto. La lógica de Hoare no nos dice que sea falso, sino que tal y como lo hemos planteado no es demostrable. Se podría plantear con otras precondiciones y poscondiciones más fuertes, estudiando el orden de ejecución de cada una de las instrucciones atómicas, y llegaríamos entonces a que es cierto, pero esta demostración es mucho más compleja.

Ejercicio 4.1.22. 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 4.1.23. 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 4.1.24. 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 4.1.25. Usando la regla de la conjunción, demostrar que

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

Aunque se podría demostrar de forma directa mediante el axioma de asignación, vamos a demostrarlo mediante la regla de la conjunción. Para ello, consideramos los siguientes triples:

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

Estos son directamente ciertos por el axioma de asignación. Por tanto, podemos aplicar la regla de la conjunción, llegando a que el siguiente triple es cierto:

$$\{i > 2\} \equiv \{V \wedge i > 2\}\ i = 2 * i\ \{i > 2 \wedge i = 2 \cdot i\} \equiv \{i > 4\}$$

Ejercicio 4.1.26. 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?

Se tiene de forma directa mediante la regla de la conjunción.

Ejercicio 4.1.27. 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 4.1.28. 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 4.1.29. 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 4.1.30. 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 4.1.31. ¿Por qué en la regla del `while B`, la condición `B` debe ser verdadera al comienzo del bucle?

Ejercicio 4.1.32. 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 4.1.33. 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 4.1.34. Demostrar la corrección del siguiente triple:

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

Distinguimos en función de los valores de i y j:

- Si $i = j$, entonces la poscondición queda $\{V\}$, luego el siguiente triple es cierto:

  $$\{V\}\ a[i] = a[i] + a[i];\ \{a[i] \geqslant a[i]\} \equiv \{V\}$$

  Por tanto, como $\{a[i] \geqslant 0\} \subset \{V\}$, se tiene que $\{a[i] \geqslant 0\} \to \{V\}$, por lo que el triple del enunciado es cierto.

- Si $i \neq j$, entonces 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 4.1.35. Verificar el siguiente segmento de programa:

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

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:

$$\{I\} \equiv \left\{ \bigwedge_{j=1}^{i-1} (a[j] = b[j]) \right\}$$
`while` $i \leqslant n$ `do begin`
$$a[i] = b[i];$$
$$i = i + 1;$$
`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 4.1.36. 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:

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

`while` $i \neq n+1$ `do begin`

$$sum = sum + f;$$

$$i = i + 1;$$

$$f = f * i;$$

`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 4.1.37. 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 4.1.38. Demostrar que para $n > 0$ el siguiente fragmento de programa termina.

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

La condición del bucle es $B = \{i \neq n\}$, y en esa condición, la única variable "variante" es `i`. El vector variante de dicha condición está formado por los valores de `i`, es decir, es:

$$\{1, 2, \ldots, n\}$$

Por tanto, nada impide que se llegue a $i = n$, y por tanto el bucle termina.

Ejercicio 4.1.39. Hallar la precondición de la terna:

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

Para ello, simplemente aplicamos el axioma de asignación, distinguiendo en función de i y j:

- Si $i \neq j$:

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

- Si $i = j$:

$$\{b = 0\} \equiv \{b = 2 * b\} \equiv \{a[i] = 2 * a[i]\}_b^{a[i]} \ a[i] = b; \ \{a[i] = 2 * a[i]\}$$

Ejercicio 4.1.40. 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 4.1.41. 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. $b = c - 2;\ a = a/b;$.

   Como no se especifica la poscondición, consideremos que esta es la más débil, $\{V\}$. Como se divide entre b, la precondición de la segunda instrucción debe ser $b \neq 0$. Es decir:

$$\{b \neq 0\}\ a = a/b;\ \{V\}$$

   Aplicando ahora el axioma de sustitución, tenemos que el siguiente triple es cierto:

$$\{c \neq 2\} \equiv \{c - 2 \neq 0\} \equiv \{b \neq 0\}^{b}_{c-2}\ b = c - 2;\ \{b \neq 0\}$$

   Por tanto, la precondición es $\{c \neq 2\}$.

Ejercicio 4.1.42. Obtener la poscondición más fuerte posible del siguiente código. Indicar todas las reglas usadas.

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

Respecto de la primera instrucción, tenemos por el axioma de asignación que:

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

Para la segunda instrucción, por el axioma de asignación, tenemos que:

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

Por tanto, por la regla de composición, tenemos que:

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

Aunque no se pide, como poscondición más débil, podemos obtener que:

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

Ejercicio 4.1.43. Verificar el siguiente código, indicando todas las reglas usadas.

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

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`:

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

Ejercicio 4.1.44. 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` (suponiendo que las posiciones van desde `1` hasta `n`), 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[1] es el siguiente:

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

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

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 usando que el siguiente triple es cierto:

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

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

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

Para comprobar su veracidad, hemos de ver que:

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

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

Como ambos se verifican, por la regla del `if` el triple anterior es cierto, por lo que (usando la regla de la composición), tenemos que $\{I \wedge B\}\ S\ \{I\}$ es cierto. Por

tanto, podemos aplicar la regla de la iteración, y obtener que:

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

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

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

$$i = i + 1;$$

$$\texttt{end}$$

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

Como hemos probado inicialmente que $\{V\}\ max = a[1];\ i = 1;\ \{I\}$, por la regla de la composición, hemos demostrado la corrección del programa.

Ejercicio 4.1.45. Demostrar la corrección parcial del siguiente código:

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

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

$$i = i + 1;$$

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

$$\texttt{end}$$

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

Para demostrar la corrección del programa, hemos de buscar un invariante global que nos permita llegar a la poscondición. Observando el código, vemos que lo que hace es almacenar en `max` el máximo del vector `a` de longitud `n` (de nuevo, suponiendo que las posiciones van desde `1` hasta `n`). Por tanto, un invariante que nos puede servir es:

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

En primer lugar, hemos de ver que el invariante es cierto al inicio del programa:

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

Esto sabemos que es cierto, ya que:

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

Posteriormente, hemos de demostrar el triple $\{I \wedge B\}\ S\ \{I\}$ con $\{B\} \equiv \{i < n\}$

y S el cuerpo del bucle para poder aplicar la regla de iteración:

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

$$i = i + 1;$$

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

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

$$\left\{ \bigwedge_{j=1}^{i} (max \geqslant a[j]) \wedge \bigvee_{j=1}^{i} (max = a[j]) \wedge i < n + 1 \right\} \rightarrow \{I\}$$

Donde hemos usado que el siguiente triple es cierto:

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

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

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

Para comprobar su veracidad, hemos de ver que:

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

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

Como ambos se verifican, por la regla del `if` el triple anterior es cierto, por lo que (usando la regla de la composición), tenemos que $\{I \wedge B\}\ S\ \{I\}$ es cierto. Por

tanto, podemos aplicar la regla de la iteración, y obtener que:

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

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

$$i = i + 1;$$

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

$$\texttt{end}$$

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

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

Como hemos probado inicialmente que $\{V\}\ max = a[1];\ i = 1;\ \{I\}$, por la regla de la composición, hemos demostrado la corrección del programa.

Ejercicio 4.1.46. 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

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

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:

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

`while` $i < n$ `do begin`

$$i = i + 1;$$

$$j = j - 1;$$

$$a[i] = b[j]$$

`end`

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

Como hemos probado inicialmente que $\{V\}\ i = 0;\ j = n;\ \{I\}$, por la regla de la composición, hemos demostrado la corrección del programa.

Ejercicio 4.1.47. Demostrar la corrección parcial del siguiente código suponiendo el invariante:

$$\{I\} \equiv \left\{ \bigwedge_{k=1}^{i-1} (a[k] = s) \wedge s = \sum_{k=1}^{i-1} A[k] \right\}$$

Donde el array A representa los valores iniciales del array a antes de ejecutar el programa.

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

Para ello, primero hemos de ver que el invariante es cierto al inicio del programa:

$$\{V\}\ i = 0;\ s = 0;\ \{I\}$$

Lo cual es cierto. A continuación:

$$\{I\} \equiv \left\{ \bigwedge_{k=1}^{i-1} (a[k] = s) \wedge s = \sum_{k=1}^{i-1} A[k] \right\}$$

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

$$\left\{ \bigwedge_{k=1}^{i-1} (a[k] = s) \wedge s = \sum_{k=1}^{i-1} A[k] \wedge i \leqslant n \right\}$$

$$s = s + a[i];$$

$$\left\{ \bigwedge_{k=1}^{i-1} (a[k] = s) \wedge s = \sum_{k=1}^{i-1} A[k] + A[i] \wedge i \leqslant n \right\} \equiv$$

$$\equiv \left\{ \bigwedge_{k=1}^{i-1} (a[k] = s) \wedge s = \sum_{k=1}^{i} A[k] \wedge i \leqslant n \right\}$$

$$a[i] = s;$$

$$\left\{ \bigwedge_{k=1}^{i} (a[k] = s) \wedge s = \sum_{k=1}^{i} A[k] \wedge i \leqslant n \right\}$$

$$i = i + 1;$$

$$\left\{ \bigwedge_{k=1}^{i-1} (a[k] = s) \wedge s = \sum_{k=1}^{i-1} A[k] \wedge i - 1 \leqslant n \right\} \rightarrow \left\{ \bigwedge_{k=1}^{i-1} (a[k] = s) \wedge s = \sum_{k=1}^{i-1} A[k] \right\} \equiv \{I\}$$

$$\texttt{end}$$

$$\left\{ \bigwedge_{k=1}^{i-1} (a[k] = s) \wedge s = \sum_{k=1}^{i-1} A[k] \wedge i > n \right\}$$

Donde en el último paso hemos aplicado la regla de la iteración: como $\{I \wedge B\}\ C\ \{I\}$ siendo $B = i \leqslant n$ y C el cuerpo del bucle, entonces se tiene que:

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

Finalmente, tendremos que:

$$\left\{ \bigwedge_{k=1}^{n} (a[k] = s) \wedge s = \sum_{k=1}^{n} A[k] \right\}$$

Ejercicio 4.1.48. Dados $i, n \geqslant 0$, $i \leqslant n$, 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);
```

Para demostrar que el código evalúa $\frac{n!}{i!(n-i)!}$, hemos de buscar un invariante global que nos permita llegar a la poscondición. Observando el código, vemos que lo que hace es calcular el factorial de `n` y almacenar en `afact` el factorial de `i` y en `bfact` el factorial de `n-i`. Por tanto, un invariante que nos puede servir es:

$$\{I\} \equiv \{fact = k! \wedge afact = (\text{mín}\{i, k\})! \wedge bfact = (\text{mín}\{n-i, k\})!\}$$

En primer lugar, hemos de ver que el invariante es cierto al inicio del programa:

$$\{0 \leqslant i \leqslant n\}\ k = 0;\ fact = 1;\ \{I\}$$

Lo cual es directamente cierto[2]. Posteriormente, hemos de demostrar el triple dado por $\{I \wedge B\}\ S\ \{I\}$ con $\{B\} \equiv \{k \neq n\}$ y S el cuerpo del bucle para poder aplicar la regla de iteración:

$$\{I \wedge B\} \equiv \{fact = k! \wedge afact = (\text{mín}\{i, k\})! \wedge bfact = (\text{mín}\{n-i, k\})! \wedge k \neq n\}$$
$$k = k + 1;$$
$$\{fact = (k-1)! \wedge afact = (\text{mín}\{i, k-1\})! \wedge bfact = (\text{mín}\{n-i, k-1\})! \wedge k \neq n+1\}$$
$$fact = fact * k;$$
$$\{fact = k! \wedge afact = (\text{mín}\{i, k-1\})! \wedge bfact = (\text{mín}\{n-i, k-1\})! \wedge k \neq n+1\}$$
$$\texttt{if}\ \ k \leqslant i\ \ \texttt{then}\ \ afact = fact;$$
$$\{fact = k! \wedge afact = (\text{mín}\{i, k\})! \wedge bfact = (\text{mín}\{n-i, k-1\})! \wedge k \neq n+1\}$$
$$\texttt{if}\ \ k \leqslant n-i\ \ \texttt{then}\ \ bfact = fact;$$
$$\{fact = k! \wedge afact = (\text{mín}\{i, k\})! \wedge bfact = (\text{mín}\{n-i, k\})! \wedge k \neq n+1\} \rightarrow \{I\}$$

donde hemos empleado dos veces la regla del `if`, veámoslo:

[2]Notemos que, para demostrar esto, hemos de suponer valores iniciales de `afact` y `bfact`, algo que no supone problema alguno.

- En primer lugar, hemos de demostrar el siguiente triple:

$$\{fact = k! \wedge afact = (\text{mín}\{i, k-1\})! \wedge bfact = (\text{mín}\{n-i, k-1\})! \wedge k \neq n+1\}$$
$$\texttt{if } k \leqslant i \texttt{ then } afact = fact;$$
$$\{fact = k! \wedge afact = (\text{mín}\{i, k\})! \wedge bfact = (\text{mín}\{n-i, k-1\})! \wedge k \neq n+1\}$$

  Para demostrarlo, hemos de demostrar los dos siguientes triples:

  - En primer lugar, hemos de demostrar que:

$$\{fact = k! \wedge afact = (\text{mín}\{i, k-1\})! \wedge bfact = (\text{mín}\{n-i, k-1\})! \wedge k \neq n+1 \wedge k \leqslant i\}$$
$$afact = fact;$$
$$\{fact = k! \wedge afact = (\text{mín}\{i, k\})! \wedge bfact = (\text{mín}\{n-i, k-1\})! \wedge k \neq n+1\}$$

    Usando la segunda regla de la consecuencia, la sustitución textual y el axioma de asignación, tenemos que:

$$\{fact = k! \wedge afact = (\text{mín}\{i, k-1\})! \wedge bfact = (\text{mín}\{n-i, k-1\})! \wedge k \neq n+1 \wedge k \leqslant i\} \rightarrow$$
$$\rightarrow \{fact = k! \wedge bfact = (\text{mín}\{n-i, k-1\})! \wedge k \neq n+1 \wedge k \leqslant i\} \equiv$$
$$\equiv \{fact = k! \wedge k! = (\text{mín}\{i, k\})! \wedge bfact = (\text{mín}\{n-i, k-1\})! \wedge k \neq n+1\} \equiv$$
$$\equiv \{fact = k! \wedge fact = (\text{mín}\{i, k\})! \wedge bfact = (\text{mín}\{n-i, k-1\})! \wedge k \neq n+1\} \equiv$$
$$\equiv \{fact = k! \wedge afact = (\text{mín}\{i, k\})! \wedge bfact = (\text{mín}\{n-i, k-1\})! \wedge k \neq n+1\}^{afact}_{fact}$$
$$afact = fact;$$
$$\{fact = k! \wedge afact = (\text{mín}\{i, k\})! \wedge bfact = (\text{mín}\{n-i, k-1\})! \wedge k \neq n+1\}$$

  - En segundo lugar, hemos de demostrar que:

$$\{fact = k! \wedge afact = (\text{mín}\{i, k-1\})! \wedge bfact = (\text{mín}\{n-i, k-1\})! \wedge k \neq n+1 \wedge k > i\}$$
$$null$$
$$\{fact = k! \wedge afact = (\text{mín}\{i, k\})! \wedge bfact = (\text{mín}\{n-i, k-1\})! \wedge k \neq n+1\}$$

    Teniendo en cuenta que $k > i \Longrightarrow k - 1 \geqslant i$, luego $\text{mín}\{i, k-1\} = i = \text{mín}\{i, k\}$, este sale directo usando el axioma de la sentencia nula y la regla de la consecuencia.

- En segundo lugar, hemos de demostrar el siguiente triple:

$$\{fact = k! \wedge afact = (\text{mín}\{i, k\})! \wedge bfact = (\text{mín}\{n-i, k-1\})! \wedge k \neq n+1\}$$
$$\texttt{if } k \leqslant n-i \texttt{ then } bfact = fact;$$
$$\{fact = k! \wedge afact = (\text{mín}\{i, k\})! \wedge bfact = (\text{mín}\{n-i, k\})! \wedge k \neq n+1\}$$

  Esta demostración es análoga a la anterior, por lo que no la repetiremos.

Habiendo demostrado que $\{I \wedge B\}\ S\ \{I\}$, podemos aplicar la regla de iteración, llegando a que:

$$\{I\} \equiv \{fact = k! \wedge afact = (\text{mín}\{i,k\})! \wedge bfact = (\text{mín}\{n-i,k\})!\}$$
$$\texttt{while}\ k <> n\ \texttt{do begin}$$
$$k = k + 1;$$
$$fact = fact * k;$$
$$\texttt{if}\ k \leqslant i\ \texttt{then}\ afact = fact;$$
$$\texttt{if}\ k \leqslant n - i\ \texttt{then}\ bfact = fact;$$
$$\texttt{end}$$
$$\{I \wedge \neg B\} \equiv \{fact = k! \wedge afact = (\text{mín}\{i,k\})! \wedge bfact = (\text{mín}\{n-i,k\})! \wedge k = n\}$$

Usando la regla de la composición (ya que hemos probado que $\{V\}\ k = 0;\ fact = 1;\ \{I\}$), y usando que $i \leqslant n$, tenemos que:

$$\{V\}$$
$$k = 0;\ fact = 1; \texttt{while}\ k <> n\ \texttt{do begin}$$
$$k = k + 1;$$
$$fact = fact * k;$$
$$\texttt{if}\ k \leqslant i\ \texttt{then}\ afact = fact;$$
$$\texttt{if}\ k \leqslant n - i\ \texttt{then}\ bfact = fact;$$
$$\texttt{end}$$
$$\{fact = k! \wedge afact = i! \wedge bfact = (n-i)!\}$$
$$bcof = fact/(afact * bfact);$$
$$\left\{bcof = \frac{n!}{i!(n-i)!}\right\}$$

donde para demostrar el último triple hemos empleado la segunda regla de la consecuencia, la sustitución textual y el axioma de asignación.

Ejercicio 4.1.49. Demostrar la terminación del fragmento de programa dado en el problema 4.1.44 ¿Qué condición se debe imponer para realizar la demostración?

En este caso, la condición del bucle es $B = \{i \neq n + 1\}$. El vector variante de dicha iteración es:

$$\{1, 2, 3, \ldots\}$$

Por tanto, para demostrar que llega al caso $i = n + 1$ (llegando así a terminar), es necesario que $n \geqslant 0$.

4.2. Sincronización en Memoria Compartida

4.2.1. Exclusión mutua

Ejercicio 4.2.1. Un algoritmo para el cual sólo pudiésemos demostrar que cumple las 4 condiciones de Dijkstra, ¿qué tipo de propiedades concurrentes satisfacería?:

(a) seguridad.

(b) vivacidad.

(c) equidad.

Justificar las respuestas.

Ejercicio 4.2.2. En algunas aplicaciones es necesario tener exclusión mutua entre procesos con la particularidad de que puede haber como mucho n procesos en una sección crítica, con n arbitrario y fijo, pero no necesariamente igual a la unidad, sino posiblemente mayor. Diseña una solución para este problema basada en el uso de espera ocupada y cerrojos. Estructura dicha solución como un par de subrutinas (usando una misma estructura de datos en memoria compartida), una para el protocolo de entrada y otro el de salida, e incluye el pseudocódigo de las mismas.

Ejercicio 4.2.3. ¿Podría pensarse que una posible solución al problema de la exclusión mutua, sería el siguiente algoritmo (de la Figura 4.9) que no necesita compartir una variable `turno` entre los 2 procesos? Demostrar (sí o no) se satisfacen las siguientes propiedades:

(a) ¿la exclusión mutua? (propiedad de seguridad)

(b) ¿la ausencia de interbloqueo? (propiedad de alcanzabilidad)

Ejercicio 4.2.4. Al siguiente algoritmo (de la Figura 4.10) se le conoce como solución de Hyman al problema de la exclusión mutua (fue publicado en una revista de impacto en 1966). ¿Es correcta dicha solución?

Ejercicio 4.2.5. Supongamos el algoritmo de exclusión mutua que expresamos a continuación. Tenemos los procesos: $0, 1, \ldots, n-1$. Cada proceso i tiene una variable $s[i]$ inicializada a 0, que puede tomar los valores 0 o 1. El proceso i puede entrar en la sección crítica si:

$$s[i] \neq s[i-1] \text{ para } i > 0$$
$$s[0] = s[n-1] \text{ para } i = 0$$

Tras ejecutar su sección crítica, el proceso i deberá hacer:

$$s[i] = s[i-1] \text{ para } i > 0$$
$$s[0] = (s[0] + 1) \mod 2 \text{ para } i = 0$$

```
{variables compartidas y valores iniciales}
var b0 : boolean := false; {true si P0 quiere acceder o esta en SC}
    b1 : boolean := false; {true si P1 quiere acceder o esta en SC}
```

```
Process P0;
begin
while true do begin
   {Protocolo de entrada}

   {indica que quiere entrar}
   b0 := true;
   {si el otro también}
   while b1 do begin
      {cede temporalmente}
      b0 := false;
      {espera}
      while b1 do begin end
      {vuelve a cerrar el paso}
      b0 := true;
   end

   {Sección crítica}
   {Protocolo de salida}
   b0 := false;
   {Resto de sentencias}
end {while}
end
```

```
Process P1;
begin
while true do begin
   {Protocolo de entrada}

   {indica que quiere entrar}
   b1 := true;
   {si el otro también}
   while b0 do begin
      {cede temporalmente}
      b1 := false;
      {espera}
      while b0 do begin end
      {vuelve a cerrar el paso}
      b1 := true;
   end

   {Sección crítica}
   {Protocolo de salida}
   b1 := false;
   {Resto de sentencias}
end {while}
end
```

Figura 4.9: Código para el Ejercicio 4.2.3.

```
{variables compartidas y valores iniciales}
var c0 : integer := 1;
    c1 : integer := 1;
    turno : integer := 1;
```

```
process P0;
begin
while true do begin
   c0 := 0;
   while turno <> 0 do begin
      while c1 = 0 do begin end
      turno := 0;
   end

   {Sección crítica}
   c0 := 1;
   {Resto de sentencias}
end
end
```

```
process P1;
begin
while true do begin
   c1 := 0;
   while turno <> 1 do begin
      while c0 = 0 do begin end
      turno := 1;
   end

   {Sección crítica}
   c1 := 1;
   {Resto de sentencias}
end
end
```

Figura 4.10: Código para el Ejercicio 4.2.4.

Ejercicio 4.2.6. Se tienen 2 procesos concurrentes que representan 2 máquinas expendedoras de tickets (en la Figura 4.11) (señalan el turno en que ha de ser atendido el cliente), los números de los tickets se representan por dos variables `n1` y `n2` que valen inicialmente 0. El proceso con el número de ticket más bajo entra en su sección crítica. En caso de tener 2 números iguales se procesa primero el proceso número 1.

(a) Demostrar que se verifica la ausencia de interbloqueo (propiedad de *alcanzabilidad* de la sección crítica), la ausencia de inanición (propiedad de *vivacidad*) y la exclusión mutua (una propiedad de *seguridad*).

(b) Demostrar que las asignaciones `n1:=1` y `n2:=1` son ambas necesarias.

Ejercicio 4.2.7. El siguiente programa (en la Figura 4.12) es una solución al problema de la exclusión mutua para 2 procesos. Discutir la corrección de esta solución: si es correcta, entonces probarlo. Si no fuese correcta, escribir escenarios que demuestren que la solución es incorrecta.

Ejercicio 4.2.8. Con respecto al algoritmo de Peterson para N procesos: ¿sería posible que llegaran 2 procesos a la etapa $N-2$, 0 procesos a la etapa $N-3$ y en todas las etapas anteriores existiera al menos 1 proceso? Justificar la respuesta.

Ejercicio 4.2.9. En el *algoritmo de Peterson* para N procesos y considerando cualquier escenario de ejecución de dicho algoritmo, el número máximo de turnos que tiene que esperar cualquier proceso para entrar en sección crítica es $N-1$ turnos.

```
{variables compartidas y valores iniciales}
var n1 : integer := 0;
    n2 : integer := 0;
```

```
Process P1;
begin
while true do begin
   n1 := 1; {E1.1}
   n1 := n2 + 1; {L1.1; E2.1}
   while n2 <> 0 and {L2.1}
         n2 < n1 do begin end;
         {L3.1}

   {Sección crítica, SC.1}
   n1 := 0; {E3.1}
   {Resto de senetencias, RS.1}
end
end
```

```
Process P2;
begin
while true do begin
   n2 := 1; {E1.2}
   n2 := n1 + 1; {L1.2; E2.2}
   while n1 <> 0 and {L2.2}
         n1 < n2 do begin end;
         {L3.2}

   {Sección crítica, SC.2}
   n2 := 0; {E3.2}
   {Resto de senetencias, RS.2}
end
end
```

Figura 4.11: Código para el Ejercicio 4.2.6.

```
{variables compartidas y valores iniciales}
var c0 : integer := 1;
    c1 : integer := 1;
```

```
Process P0;
begin
while true do begin
   repeat
      c0 := 1 - c1;
   until c1 <> 0;

   {Sección crítica}
   c0 := 1;
   {Resto de sentencias}
end
end
```

```
Process P1;
begin
while true do begin
   repeat
      c1 := 1 - c0;
   until c0 <> 0;

   {Sección crítica}
   c1 := 1;
   {Resto de sentencias}
end
end
```

Figura 4.12: Código para el Ejercicio 4.2.7.

```
1  var turno : 0..N-1;
       flag : array[0..N-1] of (pasivo, solicitando, enSC);
   flag := pasivo:

5  Process P(i);
   begin
      {Resto de instrucciones}
      repeat
         flag[i] := solicitando;
10       j := turno;
         while turno <> i do begin
            if flag[turno] = pasivo then
               turno := i;
         end
15
         flag[i] := enSC;
         j := 0;

         while j < N and ((j=i) or flag[j] != enSC) do begin
20          j := j + 1;
         end
      until (j >= N);

      {Sección crítica}
25    flag[i] := pasivo;
   end
```

Figura 4.13: Código para el Ejercicio 4.2.10.

Ejercicio 4.2.10. Con respecto al algoritmo de la Figura 4.13 (algoritmo de Dijkstra para N procesos), demostrar la falsedad de la siguiente proposición: *si un conjunto de procesos está intentando pasar simultáneamente el primer bucle* (el de la línea 11), *y el proceso que tiene el turno está pasivo, entonces siempre conseguirá entrar primero en sección crítica el proceso de dicho grupo que consiga asignar la variable turno en último lugar.*

Ejercicio 4.2.11. El algoritmo de la Figura 4.14 (algoritmo de Knuth para N procesos) resuelve el problema de la exclusión mutua para N procesos, pra lo cual utiliza N variables booleanas `flag`, una variable `turn` y la variable local `j`.

(a) Demostrar que el algoritmo de Knuth verifica todas las propiedades exigibles a un programa concurrente, incluyendo la de equidad.

(b) Escribir un escenario en el que 2 procesos consiguen pasar el bucle de la instrucción de la línea 13, suponiendo que el turno lo tiene inicialmente el proceso `p(0)`.

Ejercicio 4.2.12. Si en el algoritmo de Dijkstra de la Figura 4.13 se cambia la instrucción de la línea 12 por esta otra: `if (flag[turno] <> enSC)`, entonces el algoritmos dejaría de ser correcto. Indicar qué propiedad(es) de corrección faltaría(n) y justificar por qué.

```
var flag : array[0..N-1] of (pasivo, solicitando, enSC);
    turn := 0..N-1;
flag := pasivo;
turn := 0;

Process P(i);
var j : integer;
begin
   {Resto de instrucciones}
   repeat
      flag[i] := solicitando;
      j := turn;
      while j <> i do begin
         if flag[j] <> pasivo then
            j := turn;
         else
            j := (j - 1) mod N;
         endif;
      end

      flag[i] := enSC;
      j := 0;

      while (j < N) and ((j=i) or flag[j] != enSC) do begin
         j := j + 1;
      end
   until (j >= N);

   turn := i;
   {Seccion crítica}
   j := (turn + 1) mod N;
   turn := j;
   flag[i] := pasivo;
end
```

Figura 4.14: Código para el Ejercicio 4.2.11

Ejercicio 4.2.13. Si en el algoritmo de Knuth de la Figura 4.14 se hacen las siguientes sustituciones:

- La condición de la instrucción `until` de la línea 27 por la condición `(j >= N) and (turno = i or flag[turno] = pasivo)`.
- Se inserta el siguiente bucle después de la instrucción de la línea 31:

```
1  while (j <> turn) and (flag[j] = pasivo) do begin
       j := j + 1;
   end
```

(a) Verificar las propiedades de exclusión mutua, alcanzabilidad de la sección crítica, vivacidad y equidad del algoritmo.

(b) Calcular el número de turnos máximo que puede llegar a tener que esperar un proceso que quiera entrar en su sección crítica con el algoritmo anterior.

Ejercicio 4.2.14. Demostrar que las instrucciones entre las líneas 18 y 28 del algoritmo de exclusión mutua distribuido de Ricart-Agrawala (de la Figura 4.15) no necesitan ser protegidas dentro de la sección crítica definida por las operaciones `wait()`, `signal()` del semáforo `s`.

```
var token_presente : boolean := false;
    enSC : boolean := false;
    peticion : array[1..n] of boolean := false;
```

```
Process P(i);
begin
   wait(s);
   if not token_presente then begin
      broadcast(pet, i);
      receive(acceso);
      token_presente := true;
   end

   enSC := true;
   signal(s);

   {Sección crítica}

   enSC := false;
   wait(s);

   for j := i+1 to n, 1 to i-1 do
      if peticion[j] and
         token_presente then begin

         token_presente := false;
         send(j,acceso);
         peticion[j] := false;
      end
   end

   signal(s);
end
```

```
Process Pet(i);
begin
   receive(pet, j);
   wait(s);
   peticion[j] := true;
   if token_presente and not enSC
   then
      {Repetir líneas 18-28}
end
```

Figura 4.15: Código para el Ejercicio 4.2.14

4.2.2. Monitores

Ejercicio 4.2.15. Sean los procesos P1, P2, y P3, cuyas secuencias de instrucciones son las que se muestran en el cuadro siguiente:

```
{variables globales}
Process P1;
begin
  while true do
  begin
    a;
    b;
    c;
  end
end
```

```
Process P2;
begin
  while true do
  begin
    d;
    e;
    f;
  end
end
```

```
Process P3;
begin
  while true do
  begin
    g;
    h;
    i;
  end
end
```

Se pide resolver los siguientes problemas de sincronización, considerando que son independientes unos de otros, con semáforos. Las casuísticas son las siguientes:

1. P2 podrá pasar a ejecutar `e` sólo si P1 ha ejecutado `a` o P3 ha ejecutado `g`.

   En este caso, se trata de una espera única en la que P2 debe esperar bien a P1 o bien a P3. Podemos resolverlo usando sólo un semáforo con valor inicial 0, con lo que debemos hacer las siguientes modificaciones en los códigos superiores:

   P1:

   ```
   a;
   sem_signal(s);
   b;
   c;
   ```

   P2:

   ```
   d;
   sem_wait(s);
   e;
   f;
   ```

   P3:

   ```
   g;
   sem_signal(s);
   h;
   i;
   ```

2. P2 podrá pasar a ejecutar `e` sólo si P1 ha ejecutado `a` y P3 ha ejecutado `g`.

   Ahora, debemos resolver dos esperas únicas, ya que P2 ha de esperar tanto a P1 como ha P3. Como hay dos motivos por los que P2 ha de esperar, usaremos dos semáforos, `s1` y `s2`, ambos inicializados a 0. Las modificaciones a realizar son:

   P1:

   ```
   a;
   sem_signal(s1);
   b;
   c;
   ```

   P2:

   ```
   d;
   sem_wait(s1);
   sem_wait(s2);
   e;
   f;
   ```

   P3:

   ```
   g;
   sem_signal(s2);
   h;
   i;
   ```

3. Sólo cuando P1 haya ejecutado `b`, podrá pasar P2 a ejecutar `e` y P3 a ejecutar `h`.
   En este caso, tenemos dos procesos esperando a un mismo proceso. En esta

ocasión, tenemos que hacer uso de dos semáforos, `s1` y `s2`, ambos inicializados a 0. Notemos que no podemos hacer uso de un semáforo de forma que P2 y P3 usen `wait` y que P1 haga un solo `signal`, ya que si los dos procesos llegan al `wait` a la vez, el `signal` de P1 solo despertará a un proceso, quedándose el otro bloqueado por siempre.

Por tanto, las modificaciones a realizar son:

P1:

```
a;
b;
sem_signal(s1);
sem_signal(s2);
c;
```

P2:

```
d;
sem_wait(s1);
e;
f;
```

P3:

```
g;
sem_wait(s2);
h;
i;
```

También podríamos haber usado un sólo semáforo y que el proceso P1 realizara dos `signal` después de la instrucción `b`.

4. Sincroniza los procesos de forma que las sentencias `b` en P1, `f` en P2, y `h` en P3, sean ejecutadas como mucho por 2 procesos simultáneamente.
Se trata de una generalización de la exclusión mutua, donde en vez de ejecutar a la vez las instrucciones `b`, `f` y `h` por un solo proceso, deben ejecutarse por como mucho dos procesos a la vez. Usaremos por tanto un solo semáforo inicializado con el valor 2. Las modificaciones a realizar son:

P1:

```
a;
sem_wait(s);
b;
sem_signal(s);
c;
```

P2:

```
d;
e;
sem_wait(s);
f;
sem_signal(s);
```

P3:

```
g;
sem_wait(s);
h;
sem_signal(s);
i;
```

Ejercicio 4.2.16. El cuadro que sigue nos muestra dos procesos concurrentes, P1 y P2, que comparten una variable global `x` y las restantes variables son locales a los procesos.

```
{variables globales}
Process P1;
var m: integer;
begin
  while true do
  begin
    m:= 2*x - n;
    print(m);
  end
end
```

```

Process P2;
var d: integer;
begin
  while true do
  begin
    d:= leer_teclado();
    x:= d - c*5;
  end
end
```

Se pide:

1. Sincronizar los procesos para que P1 use todos los valores `x` suministrados por P2.

   Estamos ante un problema del estilo productor/consumidor usando como buffer intermedio una variable. Este problema ya lo aprendimos a solucionar en las prácticas, y nos basta con usar dos semáforos:

   - `sem_prod` inicializado a 1.
   - `sem_cons` inicializado a 0.

   Las modificaciones a realizar en los códigos serían las siguientes:

   P1:

```
while true do begin
  sem_wait(sem_cons);
  m:= 2*x - n;
  sem_signal(sem_prod);
  print(m);
end
```

   P2:

```
while true do begin
  d:= leer_teclado();
  sem_wait(sem_prod);
  x:= d - c*5;
  sem_signal(sem_cons);
end
```

2. Sincronizar los procesos para que P1 utilice un valor sí y otro no de la variable `x`, es decir, utilice los valores primero, tercero, quinto, etc. que vaya alcanzando dicha variable.

   Para ello, la traza del programa que nos interesa obtener es:
   `E, L, E, E, L, E, E, L, ...`

   Para ello, podemos añadir otro par de instrucciones `wait`, `signal` en el consumidor:

   P1:

```
while true do begin
  sem_wait(sem_cons);
  m:= 2*x - n;
  sem_signal(sem_prod);
  print(m);
  sem_wait(sem_cons);
  sem_signal(sem_prod);
end
```

   P2:

```
while true do begin
  d:= leer_teclado();
  sem_wait(sem_prod);
  x:= d - c*5;
  sem_signal(sem_cons);
end
```

   De esta forma, el consumidor desperdicia los valores producidos en posición par por el productor.

Ejercicio 4.2.17. Supongamos que estamos en una discoteca y resulta que está estropeado el servicio de chicas y todos tienen que compartir el de chicos. Se pretende establecer un protocolo de entrada al servicio usando semáforos que asegure siempre el cumplimiento de las siguientes restricciones:

- Chicas: sólo puede estar 1 dentro del servicio.
- Chicos: pueden entrar más de 1, pero como máximo se admitirán a 5 dentro del servicio.
- Versión machista del protocolo: los chicos tienen preferencia sobre las chicas. Esto quiere decir que si una chica está esperando entrar al servicio y llega un chico, este puede pasar y ella sigue esperando. Incluso si el chico que ha llegado no pudiera entrar inmediatamente porque ya hay 5 chicos dentro del servicio, sin embargo, pasará antes que la chica cuando salga algún chico del servicio.
- Versión feminista del protocolo: las chicas tienen preferencia sobre los chicos. Esto quiere decir que si un chico está esperando y llega una chica, ésta debe pasar antes. Incluso si la chica que ha llegado no puede entrar inmediatamente al servicio porque ya hay una chica dentro, pasará antes que el chico cuando salga la chica que está dentro.

Se pide implementar las 2 versiones del protocolo anterior utilizando semáforos POSIX. Las cabeceras que estos han de tener los semáforos no nombrados de POSIX 1003 son las siguientes:

```
// Inicialización
int sem_init(sem_t* semaforo, int pcompartido, unsigned int contador);

// Destrucción
int sem_destroy(sem_t* semaforo);

// Sincronización-espera
int sem_wait(sem_t* semaforo);

// Sincronización-señala
int sem_post(sem_t* semaforo);
```

Observación. Se han de tener en cuenta los siguientes aspectos:

1. El valor inicial del semáforo se le asigna a `contador`. Si `pcompartido` es distinto de cero, entonces el semáforo puede ser utilizado por hilos que residen en procesos diferentes; si no, sólo puede ser utilizado por hilos dentro del espacio de direcciones de un único proceso.
2. Para que se pueda destruir, el semáforo ha debido ser explícitamente inicializado mediante la operación `sem_init(...)`. La operación anterior no debe ser utilizada con semáforos nombrados.
3. Los hilos llamarán a la función `int sem_wait(sem_t* semaforo)`, pasándole un identificador de semáforo inicializado con el valor '0', para sincronizarse con una condición. Si el valor del semáforo fuera distinto de '0', entonces el valor de s se decrementa en una unidad y no bloquea.
4. La operación `int sem_post(sem_t* semaforo)` sirve para señalar a los hilos bloqueadas en un semáforo y hacer que uno pase a estar preparado para ejecutarse. Si no hay hilos bloqueados en este semáforo, entonces la ejecución

de esta operación simplemente incrementa el valor de la variable protegida (s) del semáforo. Hay que tener en cuenta que no existe ningún orden de desbloqueo definido si hay varios hilos esperando en la cola asociada a un semáforo, ya que la implementación a nivel de sistema de la operación anterior supone que el planificador puede escoger para desbloquear a cualquiera de los hilos que esperan. En particular, podría darse el siguiente escenario, otro hilo ejecutándose puede decrementar el valor del semáforo antes que cualquier hilo que vaya a ser desbloqueado como resultado de `sem_post(...)` lo pueda hacer y, posteriormente, se volvería a bloquear el hilo despertado.

Versión machista.

Para plantear la solución en código, hemos creado 4 funciones: `entra_chico`, `entra_chica`, `sale_chico` y `sale_chica`, con el fin de simular el problema planteado.

En estas funciones, debemos usar unas variables compartidas que nos vayan indicando cuántos chicas y chicos hay esperando, así como si el baño está ocupado por chicos o por una chica.

Mostramos ahora las variables compartidas a declarar, junto con su código de inicialización y el código que debemos usar tras el cierre de los servicios para destruir los semáforos:

```
1  const int MAX = 5;
   int chicos_esperando, chicas_esperando, chicos_dentro;
   bool chica_dentro;
   sem_t* mutex, max, cola_chicos, cola_chicas;
5
   void inicializacion(){
      chicos_esperando = chicas_esperando = chicos_dentro = 0;
      chica_dentro = false;
      sem_init(mutex, 1, 1);
10    sem_init(max, 1, MAX);
      sem_init(cola_chicos, 1, 0);
      sem_init(cola_chicas, 1, 0);
   }

15 void destruccion(){
      sem_destroy(mutex);
      sem_destroy(max);
      sem_destroy(cola_chicos);
      sem_destroy(cola_chicas);
20 }
```

A continuación, las funciones de entrada y salida del baño para los chicos:

```
1  void entra_chico(){
      sem_wait(max);   // Espera a que salga uno
      sem_wait(mutex); // Adquiere ex. mutua (em)
```

```
    if(chica_dentro){    // Baño ocupado
        chicos_esperando++;
        sem_post(mutex);    // Libera em
        sem_wait(cola_chicos);
        chicos_esperando--;
    }

    chicos_dentro++;
    // Si puede entrar otro
    if(chicos_dentro < MAX && chicos_esperando > 0){
        sem_post(cola_chicos);
    }else{
        sem_post(mutex);
    }
}

void sale_chico(){
    sem_wait(mutex);     // Adquiere em
    chicos_dentro--;

    // Si se queda el baño libre, hay chica esperando pero no chicos
    if(chicos_esperando == 0 && chicas_esperando > 0 && chicos_dentro ==
    0){
        sem_post(cola_chicas);
    }else{
        sem_post(mutex);
    }
    sem_post(max);
}
```

Ahora, las funciones de entrada y salida de las chicas serían:

```
void entra_chica(){
    sem_wait(mutex);     // Adquiere em

    // Si el baño está ocupado o hay un chico esperando
    if(chicos_dentro > 0 || chica_dentro || chicos_esperando > 0){
        chicas_esperando++;
        sem_post(mutex);
        sem_wait(cola_chicas);
        chicas_esperando--;
    }

    chica_dentro = true;
    sem_post(mutex);
}

void sale_chica(){
    sem_wait(mutex);     // Adquiere em
    chica_dentro = false;

    if(chicos_esperando > 0){    // Prioridad a chicos
        sem_post(cola_chicos);
    }else if(chicas_esperando > 0){
        sem_post(cola_chicas);
    }else{
```

```
25          sem_post(mutex);
        }
    }
```

Versión feminista.

Para esta versión, usaremos las mismas variables compartidas con las mismas funciones `inicializacion` y `destruccion` de la versión anterior.

A continuación, las funciones de entrada y salida del baño para los chicos:

```
1   void entra_chico(){
        sem_wait(max);   // Espera a que salga un chico del baño
        sem_wait(mutex); // Para ex. mutua (em)

5       if(chicas_esperando > 0 || chica_dentro){    // Prioridad a chicas
            chicos_esperando++;   // Un nuevo chico esperando
            sem_post(mutex);    // Libera em antes de bloquearse
            sem_wait(cola_chicos);

10          chicos_esperando--;
        }

        // Ya no hay chicas en cola o dentro del baño

15      chicos_dentro++;
        if(chicos_cola > 0 && chicos_dentro < MAX){  // Si puede entrar otro
            sem_post(cola_chicos);
        }else{
            sem_post(mutex);
20      }
    }

    void sale_chico(){
        sem_wait(mutex);     // Espera para em
25      chicos_dentro--;

        // Si hay una chica y puede entrar
        if(chicos_dentro == 0 && chicas_esperando > 0){
            sem_post(cola_chicas);
30      }else{
            sem_post(mutex);  // Libera em
        }

        sem_post(max);       // Un chico menos
35  }
```

Ahora, mostramos el código de entrada y salida para las chicas:

```
1   void entra_chica(){
        sem_wait(mutex);     // Adquiere em

        if(chicos_dentro > 0 || chica_dentro){   // Si baño ocupado
5           chicas_esperando++;
            sem_post(mutex);
```

```
        sem_wait(cola_chicas);

        chicas_esperando--;
    }

    chica_dentro = true;
    sem_post(mutex);
}

void sale_chica(){
    sem_wait(mutex);      // Adquiere em
    chica_dentro = false;

    if(chicas_esperando > 0){    // Desbloquea a chica
        sem_post(cola_chicas);
    }else if(chicos_esperando > 0){  // Desbloquea a chico
        sem_post(cola_chicos);
    }else{
        sem_signal(mutex);
    }
}
```

Ejercicio 4.2.18. Aunque un monitor garantiza la exclusión mutua, los procedimientos tienen que ser reentrantes. Explicar por qué.

Los procedimientos han de ser reentrantes porque queremos tener procesos que ejecuten procedimientos del monitor, que estos puedan bloquearse durante la ejecución de los mismos y que puedan desbloquearse tras unas condiciones y que puedan seguir ejecutando la función por donde iban.

Es necesario que los procedimientos sean reentrantes para que los procesos que ejecutan los procedimientos y que se bloquean sean capaces de seguir la ejecución de la función una vez desbloqueados por la instrucción por la que se quedaron, manteniendo intactos los valores de las variables locales usadas en el procedimiento.

Ejercicio 4.2.19. Se consideran dos tipos de recursos accesibles por varios procesos concurrentes (denominamos a los recursos como recursos de tipo 1 y de tipo 2). Existen N_1 ejemplares de recursos de tipo 1 y N_2 ejemplares de recursos de tipo 2. Para la gestión de estos ejemplares, queremos diseñar un monitor (con semántica SU) que exporta un procedimiento (`pedir_recurso`), para pedir un ejemplar de uno de los dos tipos de recursos. Este procedimiento incluye un parámetro entero (`tipo`), que valdrá 1 ó 2 indicando el tipo del ejemplar que se desea usar, así mismo, el monitor incorpora otro procedimiento (`liberar_recurso`) para indicar que se deja de usar un ejemplar de un recurso previamente solicitado (este procedimiento también admite un entero que puede valer 1 ó 2, según el tipo de ejemplar que se quiera liberar). En ningún momento puede haber un ejemplar de un tipo de recurso en uso por más de un proceso.

En este contexto, responde a estas cuestiones:

1. Implementa el monitor con los dos procedimientos citados, suponiendo que N_1 y N_2 son dos constantes arbitrarias, mayores que cero.

2. El uso de este monitor puede dar lugar a interbloqueo. Esto ocurre cuando más de un proceso, en algún punto en su código, tiene la necesidad de usar dos ejemplares de recursos de distinto tipo a la vez. Describe la secuencia de peticiones (llamadas al procedimiento correspondiente del monitor) que da lugar a interbloqueo.

3. Una posible solución al problema anterior es obligar a que si un proceso necesita dos recursos de distinto tipo a la vez, deba de llamar a `pedir_recurso`, dando un parámetro con valor 0, para indicar que necesita los dos ejemplares. En esta solución, cuando un ejemplar quede libre, se dará prioridad a los posibles procesos esperando usar dos ejemplares, frente a los que esperan usar solo uno de ellos.

1. Mostramos la implementación usando pseudocódigo:

```
monitor Recursos (N1, N2 : integer);
   var libres : array[1..2] of integer;
       colas : array[1..2] of condition;

   begin
      libres[1] = N1;
      libres[2] = N2;
   end

   procedure pedir_recurso(tipo : 1..2);
   begin
      if libres[tipo] = 0 then
         colas[tipo].wait();
      end;
      libres[tipo]--;
   end

   procedure liberar_recurso(tipo : 1..2);
   begin
      libres[tipo]++;
      cola[tipo].signal();
   end
end
```

2. Para mostrar una situación en la que el uso de este monitor puede dar lugar a un interbloqueo, mostramos los siguientes códigos:

```
var monitor : Recursos(1,1);

process P1;
begin
   monitor.pedir_recurso(1);
   monitor.pedir_recurso(2);
   {Uso de los recursos}
   monitor.liberar_recurso(1);
   monitor.liberar_recurso(2);
end
```

```
process P2;
begin
   monitor.pedir_recurso(2);
   monitor.pedir_recurso(1);
   {Uso de los recursos}
   monitor.liberar_recurso(2);
   monitor.liberar_recurso(1);
end

begin
   cobegin P1 || P2 coend
end
```

Si ahora se ejecutan los códigos y se sucede un entrelazamiento en las instrucciones del programa concurrente obteniendo la traza:

P1:	pedir_recurso(1)
P2:	pedir_recurso(2)
P1:	pedir_recurso(2)
P2:	pedir_recurso(1)

Después de las dos primeras instrucciones, no quedarán recursos de ningún tipo libres, pero P1 no podrá ejecutarse ya que está esperando a un recurso de tipo 2 y P2 tampoco podrá hacerlo, al estar esperando a un recurso de tipo 1.

Notemos que con monitores podemos demostrar la corrección parcial de los programas, pero no que estos terminen (esto es, que estén libres de interbloqueos o situaciones similares). Es responsabilidad del programador evitar este tipo de situaciones.

3. Aunque no podemos crear un monitor que evite el interbloqueo en un mal uso del monitor, sí que podemos ofrecer una solución de compromiso que ayude al programador a evitar este tipo de situaciones, tal y como se describe en el enunciado y mostramos con el siguiente código:

```
    monitor Recursos (N1, N2 : integer);
    var libres : array[1..2] of integer;
        colas : array[0..2] of condition;

    begin
       libres[1] := N1;
       libres[2] := N2;
    end

    procedure pedir_recurso(tipo : 0..2);
    begin
       if tipo = 0 then
          if(libres[1] = 0 or libres[2] = 0) then
             colas[0].wait();
          end
          libres[1]--;
          libres[2]--;
```

```
        else begin
           if(libres[tipo] = 0) then
              colas[tipo].wait();
           end
           libres[tipo]--;
        end
     end

     procedure liberar_recurso(tipo : 1..2);
     var otro_tipo = tipo mod 2 + 1;
     begin
        libres[tipo]++;

        {Si hay otro tipo}
        if(libres[otro_tipo] > 0 and cola[0].queue()) then
           cola[0].signal();
        else
           cola[tipo].signal();
        end
     end
end
```

Ejercicio 4.2.20. Escribir una solución al problema de lectores-escritores con monitores:

1. Con prioridad a los lectores: quiere decir que, si en un momento puede acceder al recurso, tanto un lector como un escritor, se da paso preferentemente al lector.

2. Con prioridad a los escritores: quiere decir que, si en un momento puede acceder tanto un lector como un escritor, se da paso preferentemente al escritor.

3. Con prioridades iguales: en este caso, los procesos acceden al recurso estrictamente en orden de llegada, lo cual implica, en particular, que si hay lectores leyendo y un escritor esperando, los lectores que intenten acceder después del escritor no podrán hacerlo hasta que no lo haga dicho escritor.

En este problema, contamos con procesos de dos tipos, lectores y escritores. Si un escritor hace uso del recurso compartido, este debe hacerlo en exclusión mutua. Sin embargo, si un lector hace uso del recurso, puede haber más lectores que también lo estén usando al mismo tiempo. Planteamos ahora la solución a cada uno de los puntos, usando para ello 2 procedimientos por cada tipo de proceso que interviene en el problema (un procedimiento de entrada al recurso y otro de fin de uso):

1. Solución para dar prioridad a los lectores:

```
Monitor LecEsc;
   var lec_dentro : integer;
       esc_dentro : boolean;
       cola_lec, cola_esc : condition;

   begin
```

```
      lec_dentro = 0;
      esc_dentro = false;
   end

   procedure entra_lector();
   begin
      if esc_dentro then
         cola_lec.wait();
      end

      lec_dentro++;

      if cola_lec.queue() then  {por eficiencia}
         cola_lec.signal();
      end
   end

   procedure sale_lector();
   begin
      lec_dentro--;

      if lec_dentro = 0 then
         cola_esc.signal();
      end
   end

   procedure entra_escritor();
   begin
      if lec_dentro > o OR esc_dentro then
         cola_esc.wait();
      end

      esc_dentro = true;
   end

   procedure sale_escritor();
   begin
      esc_dentro = false;

      if cola_lec.queue() then
         cola_lec.signal();
      else
         cola_esc.signal();
      end
   end
end
```

2. Solución para dar prioridad a los esctirores. En este caso, el código es idéntico, salvo en el procedimiento `sale_escritor`, que quedaría de la siguiente forma:

```
procedure sale_escritor();
begin
   esc_dentro = false;
```

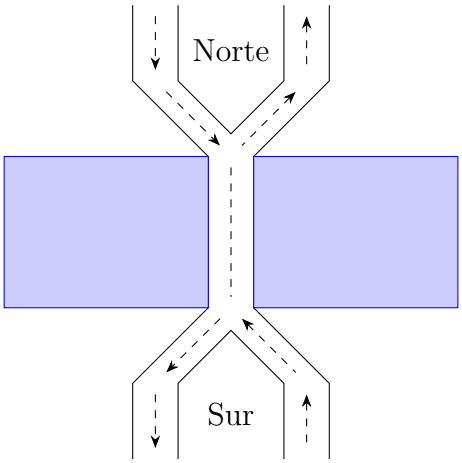


Figura 4.16: Problema de exclusión mutua en el acceso de coches desde 2 sentidos opuestos a un puente de un solo carril.

```
5     if cola_esc.queue() then
          cola_esc.signal();
      else
          cola_lec.signal();
      end
10 end
```

3. Finalmente, para dar una solución en la que lectores y escritores tienen prioridades iguales.

Ejercicio 4.2.21. Varios coches que vienen del norte y del sur pretenden cruzar un puente sobre un río (ver Figura 4.16). Sólo existe un carril sobre dicho puente. Por lo tanto, en un momento dado, el puente solo puede ser cruzado por uno o más coches en la misma dirección (pero no en direcciones opuestas).

1. Completar el código del siguiente monitor que resuelve el problema del acceso al puente suponiendo que llega un coche del norte (sur) y cruza el puente si no hay otro coche del sur (norte) cruzando el puente en ese momento.

```
1  Monitor Puente
   var ... ;
   procedure EntrarCocheDelNorte()
   begin
5  ...
   end
   procedure SalirCocheDelNorte()
   begin
   ....
10 end
   procedure EntrarCocheDelSur()
   begin
   ....
   end
```

```
15 procedure SalirCocheDelSur()
   begin
   ...
   end
   { Inicializacion }
20 begin
   ....
   end
```

2. Mejorar el monitor anterior, de forma que la dirección del tráfico a través del puente cambie cada vez que lo hayan cruzado 10 coches en una dirección, mientras 1 ó más coches estuviesen esperando cruzar el puente en dirección opuesta.

1. Este problema es una simplificación del problema de los lectores/escritores. Para el mismo, proponemos el siguiente monitor como solución al problema:

```
1  Monitor Puente;
       var coches_N, coches_S : integer;
           cola_N, cola_S : condition;

5      procedure EntrarCocheDelNorte();
       begin
          { Hay coches en otro sentido }
          if coches_S > 0 then
             cola_N.wait();
10
          coches_N := coches_N + 1;

          { Por temas de eficiencia }
          if coches_N.queue() then
15           coches_N.signal();
       end

       procedure SalirCocheDelNorte();
       begin
20        coches_N := coches_N - 1;

          { El puente se queda vacío }
          if coches_N = 0 then
             coches_S.signal();
25     end
```

```
1      procedure EntrarCocheDelSur();
       begin
          { Hay coches en otro sentido }
          if coches_N > 0 then
5            cola_S.wait();

          coches_S := coches_S + 1;

          { Por temas de eficiencia }
          if coches_S.queue() then
10           coches_S.signal();
       end

       procedure SalirCocheDelSur();
       begin
15        coches_S := coches_S - 1;

          { El puente se queda vacío }
          if coches_S = 0 then
             coches_N.signal();
20     end

       begin
          coches_N := 0; coches_S := 0;
       end
25  end
```

2. Ahora, una vez que haya algún coche en sentido opuesto, contabilizaremos el número de coches que pasan en nuestro sentido mientras que hay esperando en sentido opuesto, con la finalidad de invertir el sentido una vez hayan pasado `umbral` coches:

```
Monitor Puente;
   var coches_N, coches_S, N_pueden, S_pueden : integer;
       cola_N, cola_S : condition;

   procedure EntrarCocheDelNorte();
   begin
      { Hay coches en otro sentido }
      if (coches_S > 0 or N_pueden = 0) then cola_N.wait();

      coches_N := coches_N + 1;

      { Si hay esperando en el sur }
      if cola_S.queue() then N_pueden := N_pueden - 1;

      if (coches_N.queue() and N_pueden > 0) then coches_N.signal();
   end

   procedure SalirCocheDelNorte();
   begin
      coches_N := coches_N - 1;

      { El puente se queda vacío }
      if coches_N = 0 then
      begin
         S_pueden := UMBRAL;
         coches_S.signal();
      end
   end

   procedure EntrarCocheDelSur();
   begin
      { Hay coches en otro sentido }
      if (coches_N > 0 or S_pueden = 0) then cola_S.wait();

      coches_S := coches_S + 1;

      { Si hay esperando en el norte }
      if cola_N.queue() then S_pueden := S_pueden - 1;

      if (coches_S.queue() and S_pueden > 0) then coches_S.signal();
   end

   procedure SalirCocheDelSur();
   begin
      coches_S := coches_S - 1;

      { El puente se queda vacío }
      if coches_S = 0 then
      begin
         N_pueden = UMBRAL;
         coches_N.signal();
      end
   end

   begin
      coches_N := 0; coches_S := 0; N_pueden := UMBRAL; S_pueden := UMBRAL;
   end
end
```

Ejercicio 4.2.22. Una tribu de antropófagos comparte una olla en la que caben M misioneros. Cuando algún salvaje quiere comer, se sirve directamente de la olla, a no ser que ésta esté vacía. Si la olla está vacía, el salvaje despertará al cocinero y esperará a que éste haya rellenado la olla con otros M misioneros. Para solucionar la sincronización usamos un monitor llamado `Olla`, que se puede usar así:

```
monitor Olla ;
....
begin
....
end;
proceso ProcSalvaje[ i:1..N ] ;
begin
while true do begin
        Olla.Servirse_1_misionero();
        Comer(); { es un retraso aleatorio }
    end
end;
proceso ProcCocinero ;
begin
while true do begin
        Olla.Dormir();
        Olla.Rellenar_Olla();
    end
end;
```

Se pide diseñar el código del monitor `Olla` para que se satisfaga la sincronización requerida en el enunciado del problema, teniendo en cuenta que:

- La solución propuesta no debe producir interbloqueos.
- Los salvajes podrán comer siempre que haya comida en la olla.
- Sólo se ha de despertar al proceso cocinero cuando la olla esté vacía.

La solución al problema viene dada gracias al monitor (donde `M` es una constante que fija el número de misioneros a rellenar cada vez):

```
Monitor Olla;
   var num_misioneros : integer;
       comer, dormir : condition;

   procedure Servirse_1_misionero();
   begin
      { Si no quedan misioneros }
      if num_misioneros = 0 then
      begin
         dormir.signal();  { Despierta al cocinero }
         comer.wait();
      end

      num_misioneros := num_misioneros - 1;

      if num_misioneros > 0 then
         comer.signal();
   end

   procedure Dormir();
   begin
      if num_misioneros > 0 then
         dormir.wait();
   end

   procedure Rellenar_Olla();
   begin
      num_misioneros := M;

      { Avisa a un salvaje }
      comer.signal();
   end

   begin
      num_misioneros := M;
   end
end
```

Ejercicio 4.2.23. Una cuenta de ahorros es compartida por varias personas (procesos). Cada persona puede depositar o retirar fondos de la cuenta. El saldo actual de la cuenta es la suma de todos los depósitos menos la suma de todos los reintegros. El saldo nunca puede ser negativo. Queremos usar un monitor para resolver el problema.

El monitor debe tener 2 procedimientos: `depositar(c)` y `retirar(c)`. Suponer que los argumentos de las 2 operaciones son siempre positivos, e indican las cantidades a depositar o retirar. El monitor usará la semántica señalar y espera urgente (SU). Se deben de escribir varias versiones de la solución, según las variaciones de los requerimientos que se describen a continuación:

1. Todo proceso puede retirar fondos mientras la cantidad solicitada `c` sea menor o igual que el saldo disponible en la cuenta en ese momento. Si un proceso intenta retirar una cantidad `c` mayor que el saldo, debe quedar bloqueado hasta que el saldo se incremente lo suficiente (como consecuencia de que otros

procesos depositen fondos en la cuenta) para que se pueda atender la petición. Hacer dos versiones del monitor:

a) Colas normales FIFO sin prioridad.

b) Con colas de prioridad.

2. El reintegro de fondos a los clientes se hace únicamente según el orden de llegada, si hay más de un cliente esperando, sólo el primero que llegó puede optar a retirar la cantidad que desea, mientras esto no sea posible, esperarán todos los clientes, independientemente de cuanto quieran retirar los demás. Por ejemplo, suponer que el saldo es 200 unidades y un cliente está esperando un reintegro de 300 unidades, entonces si llega otro cliente debe esperarse, incluso si quiere retirar 200 unidades. De nuevo, resolverlo utilizando dos versiones:

   a) Colas normales FIFO sin prioridad.

   b) Con colas de prioridad.

1. Para la primera versión:

   a) Con colas normales FIFO sin prioridad en las variables condición:

```
Monitor Cuenta;
var saldo : integer;
    cola : condition;

   procedure depositar(c : integer);
   begin
      saldo = saldo + c;
      cola.signal();
   end

   procedure retirar(c : integer);
   begin
      while c > saldo do begin
         cola.signal();
         cola.wait();
      end do
      saldo = saldo - c;
      cola.signal();
   end

   begin
      saldo = 0;
   end
end
```

   b) Con colas de prioridad en las variables condición, el código del monitor queda más simple, teniendo solo que modificar el procedimiento `retirar`:

```
procedure retirar(c : integer);
begin
   while c > saldo do begin
      cola.wait(c);
   end do
   saldo = saldo - c;
   cola.signal();
end
```

2. Ahora, con orden en las retiradas, es necesrio disponer de una variable condición más, que controle si hay alguien esperando a retirar dinero.

   a) Con colas normales FIFO en las variables condición:

```
Monitor Cuenta;
var saldo : integer;
    cola, ventanilla : condition;

   procedure depositar(c : integer);
   begin
      saldo = saldo + c;
      cola.signal();
   end

   procedure retirar(c : integer);
   begin
      if ventanilla.queue() then
         cola.wait();

      while c > saldo do begin
         ventanilla.wait();
      end do

      saldo = saldo - c;
      cola.signal();
   end

   begin
      saldo = 0;
   end
end
```

   b) Con colas con prioridad en las variables condición, lo que hacemos es introducir una nueva variable `contador`, con la finalidad de bloquear a los procesos según su orden de llegada.

```
Monitor Cuenta;
var saldo : integer;
    contador : integer := 0;
    cola : condition;

   procedure depositar(c : integer);
   begin
```

```
      saldo = saldo + c;
      cola.signal();
   end

   procedure retirar(c : integer);
   var ticket : integer;
   begin
      ticket = contador;
      contador = contador + 1;

      if cola.queue() then
         cola.wait(ticket);

      while c > saldo do
         cola.wait(ticket);
      end

      saldo = saldo - c;
      cola.signal();
   end

   begin
      saldo = 0;
   end
end
```

Ejercicio 4.2.24. Los procesos P1, P2, ..., Pn comparten un único recurso R, pero sólo un proceso puede utilizarlo cada vez. Un proceso Pi puede comenzar a utilizar R si está libre; en caso contrario, el proceso debe esperar a que el recurso sea liberado por otro proceso. Si hay varios procesos esperando a que quede libre R, se concederá al proceso que tenga mayor prioridad. La regla de prioridad de los procesos es la siguiente: el proceso Pi tiene prioridad i, (con $1 \leqslant i \leqslant n$), donde los números menores implican mayor prioridad (es decir, si $i < j$, entonces Pi pasa por delante de Pj). Implementar un monitor que implemente los procedimientos `Pedir(...)` y `Liberar()` con un monitor que garantice la exclusión mutua y el acceso prioritario del procesos al recurso R.

Suponiendo que podemos usar variables condición con colas prioritarias, la solución al ejercicio es:

```
Monitor Recurso;
var libre : boolean;
    cola : condition;

   begin
      libre = true;
   end

   procedure Liberar();
   begin
      libre = true;
      cola.signal();
   end
```

```
   procedure Pedir(id : integer); {Nº de proceso}
   begin
      if not libre then
         cola.wait(id);
      end

      libre = false;
   end
end
```

Ejercicio 4.2.25. El siguiente monitor (`Barrera2`) proporciona un único procedimiento de nombre `entrada()`, que provoca que el primer proceso que lo llama sea suspendido y el segundo que lo llama despierte al primero que lo llamó (a continuación ambos continúan), y así actúa cíclicamente. Obtener una implementación de este monitor usando semáforos.

```
Monitor Barrera2 ;
var n : integer; { num. de proc. que han llegado desde el signal }
s : condicion ; { cola donde espera el segundo }
procedure entrada() ;
    begin
        n := n+1 ; { ha llegado un proceso mas }
        if n < 2 then { si es el primero: }
            s.wait() { esperar al segundo }
        else begin { si es el segundo: }
            n := 0; { inicializa el contador }
            s.signal() { despertar al primero }
        end
    end

    { Inicializacion }
    begin
        n := 0 ;
    end
end
```

Para realizar esta implementación, haremos uso de las siguientes variables compartidas:

```
var n : integer;
    mutex, sem : semaphore;
```

donde inicializaremos los semáforos `mutex` y `sem` a 0. Usaremos la siguiente función, que hará las veces del procedimiento `entrada` del monitor:

```
procedure entrada();
begin
   sem_wait(mutex);  {Adquiere em}
   n := n + 1;

   if n < 2 then begin {Si era el primero}
      sem_signal(mutex);  {Libera em antes de bloquearse}
      sem_wait(sem);
```

```
   else then begin {Si era el segundo}
      n := 0;
      sem_signal(sem);
      sem_signal(mutex);
   end
end
```

Ejercicio 4.2.26. Este es un ejemplo clásico que ilustra el problema del interbloqueo, y aparece en la literatura informática con el nombre de el problema de los filósofos-comensales. Se puede enunciar como se indica a continuación: sentados a una mesa están cinco filósofos, la actividad de cada filósofo es un ciclo sin fin de las operaciones de pensar y comer; entre cada dos filósofos hay un tenedor y para poder comer, un filósofo necesita obligatoriamente dos tenedores: el de su derecha y el de su izquierda. Se han definido cinco procesos concurrentes, cada uno de ellos describe la actividad de un filósofo. Los procesos usan un monitor, llamado `MonFilo`. Antes de comer cada filósofo debe disponer de su tenedor de la derecha y el de la izquierda, y cuando termina la actividad de comer, libera ambos tenedores. El filósofo i alude al tenedor de su derecha como el número i, y al de su izquierda como el número $i + 1 \mathrm{mod} 5$. El monitor `MonFilo` exportará dos procedimentos: `coge_tenedor(num_tenedor,num_proceso)` y `libera_tenedor(num_tenedor)` para indicar que un proceso filósofo desea coger un tenedor determinado. El código del programa (sin incluir la implementación del monitor) es el siguiente:

```
monitor MonFilo ;
....
procedure coge_tenedor( num_ten, num_proc : integer );
....
procedure libera_tenedor( num_ten : integer );
....
begin
....
end
proceso Filosofo[ i: 0..4 ] ;
    begin
        while true do begin
            MonFilo.coge_tenedor(i,i);          {argumento 1=codigo tenedor}
            MonFilo.coge_tenedor(i+1 mod 5,i);  {argumento 2=numero de proceso}
        comer();
        MonFilo.libera_tenedor(i);
        MonFilo.libera_tenedor(i+1 mod 5);
        pensar();
    end
end
```

Con este interfaz para el monitor, responde a las siguientes cuestiones:

1. Diseña una solución para el monitor `MonFilo`.

2. Describe la situación de interbloqueo que puede ocurrir con la solución que has escrito antes.

3. Diseña una nueva solución, en la cual se evite el interbloqueo descrito, para ello, esta solución no debe permitir que haya más de cuatro filósofos simultáneamente intentado coger su primer tenedor.

1. Hemos diseñado el siguiente monitor, donde `tenedores[i]` indica si el tenedor i-ésimo ha sido (`true`) cogido o si no (`false`).

```
monitor MonFilo;
   var tenedores : array[0..4] of boolean;
       filosofo : array[0..4] of condition;

   procedure coge_tenedor(num_tenedor, num_proceso : integer);
   begin
      { Si ya han cogido el tenedor }
      if tenedores[num_tenedor] then
         filosofo[num_proceso].wait();

      tenedores[num_tenedor] := true;
   end

   procedure libera_tenedor(num_tenedor);
   begin
      tenedores[num_tenedor] := false;

      {Si el filósofo de la izqda estaba esperando se despierta}
      if filosofo[num_tenedor].queue() then
         filosofo[num_tenedor].signal();
      else then  {Si no, despertamos al filósofo de la dcha}
         filosofo[(num_tenedor - 1) mod 5].signal();
   end

   begin
      for i = 0 to 4 do
         tenedores[i] := false;
      end
   end
end
```

2. Ante el código proporcionado para los filósofos anteriormente, puede producirse una situación de interbloqueo si se sucede la siguiente traza de ejecución:

	Nº Proceso	Llamada a procedimiento
1	0	`MonFilo.coge_tenedor(0,0);`
2	1	`MonFilo.coge_tenedor(1,1);`
3	2	`MonFilo.coge_tenedor(2,2);`
4	3	`MonFilo.coge_tenedor(3,3);`
5	4	`MonFilo.coge_tenedor(4,4);`

Tabla 4.2: Traza de ejecución que lleva a una situación de interbloqueo.

En este instante, todos los filósofos han cogido el tenedor de su derecha, el cual no soltarán hasta comer, para lo que necesitan el tenedor de su izquierda.

Sin embargo, como son 5 filósofos y solo hay disponibles 5 tenedores, ningún filósofo conseguirá nunca su tenedor de la izquierda para poder comer. De esta forma, todos los filósofos quedarán bloqueados por siempre.