

Sistemas Concurrentes y Distribuidos



*Escuela Técnica Superior de Ingenierías
Informática y de Telecomunicación*

Los Del DGIIM, losdeldgiim.github.io

Doble Grado en Ingeniería Informática y Matemáticas
Universidad de Granada



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

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

Sistemas Concurrentes y Distribuidos

Los Del DGIIM, losdeldgiim.github.io

José Juan Urrutia Milán

Granada, 2024-2025

Índice general

1. Introducción a la Programación Concurrente	7
1.1. Conceptos básicos	7
1.1.1. Comparación de programas concurrentes con secuenciales . . .	7
1.1.2. Definición de concurrencia	8
1.1.3. Axiomas de la programación concurrente	8
1.2. Modelos para creación de procesos en un programa	10
1.2.1. Grafos de Sincronización	10
1.2.2. Definición estructurada de procesos	11
1.2.3. Definición no estructurada de procesos	12
1.3. Exclusión mutua y sincronización	12
1.4. Propiedades de los sistemas concurrentes	15
1.4.1. Propiedades de seguridad (<i>safety</i>)	15
1.4.2. Propiedades de vivacidad (<i>liveness</i>)	15
1.5. Lógica de programas de Hoare y verificación de programas concurrentes	17
1.5.1. Corrección de los programas concurrentes	17
1.5.2. Introducción a la Lógica de Hoare	18
1.5.3. Lógica de programas	20
1.5.4. Verificación de sentencias concurrentes	22
1.5.5. Verificación usando invariantes globales	29
1.5.6. Demostrar que un programa termina	31
2. Sincronización en memoria compartida. Monitores	33
2.1. Problema de la exclusión mutua	33
2.1.1. Condiciones de Dijkstra	34
2.1.2. Método de refinamiento sucesivo	34
2.1.3. Algoritmo de Dekker	37
2.1.4. Algoritmo de Dijkstra	41
2.1.5. Algoritmo de Knuth	44
2.1.6. Algoritmo de Peterson para 2 procesos	50
2.1.7. Algoritmo de Peterson para n procesos	52
2.2. Definición de un monitor	60
2.2.1. Concepto de monitor	61
2.2.2. Características de programación con monitores	64
2.2.3. Exclusión mutua en los procedimientos de un monitor	66
2.2.4. Operaciones de sincronización	67
2.3. Verificación de programas con monitores	70
2.3.1. Invariante de monitor	71

2.3.2.	Axiomas para operaciones de sincronización desplazantes . . .	72
2.3.3.	Regla de concurrencia para programas con monitores	79
2.4.	Patrones de uso de un monitor	80
2.4.1.	Espera única	80
2.4.2.	Exclusión mutua	82
2.4.3.	Productores/Consumidores	84
2.5.	Semánticas de señales	86
2.5.1.	Señalar y Continuar (SC)	86
2.5.2.	Señalar y Salir (SS)	88
2.5.3.	Señalar y Esperar (SE)	88
2.5.4.	Señalar y Espera Urgente (SU)	89
2.5.5.	Comparativa	91
2.5.6.	Axiomas para operaciones de sincronización no desplazantes .	95
2.5.7.	Intercambio de señales en programas que usan monitores . . .	95
2.5.8.	Señales <code>wait</code> con prioridad	96
2.6.	Implementación de los monitores	98
3.	Sistemas basados en paso de mensajes	101
3.1.	Introducción a la programación distribuida	101
3.1.1.	Multiprocesamiento	102
3.1.2.	Multiprogramación SPMD	104
3.2.	Semántica de las operaciones de paso de mensajes	105
3.2.1.	Operaciones bloqueantes	106
3.2.2.	Operaciones no bloqueantes	109
3.3.	Diseño de programas distribuidos	110
3.3.1.	Tipos de procesos	110
3.3.2.	Órdenes con guarda	111
3.4.	Espera selectiva	112
3.4.1.	Definición de sentencia de espera selectiva o <code>select</code>	113
3.4.2.	Comportamiento de la espera selectiva	114
3.4.3.	<code>select</code> con guardas indexadas	116
3.4.4.	<code>select</code> con sentencia <code>else</code>	117
4.	Sistemas de Tiempo Real	119
4.1.	Consideraciones sobre el Tiempo	120
4.1.1.	Relojes de tiempo real	121
4.1.2.	Temporizadores y retardos	121
4.2.	Modelo simple de tareas	122
4.2.1.	Tipos de tareas de tiempo real	122
4.2.2.	Características del modelo simple de tareas	123
4.3.	Atributos temporales de una tarea	124
4.4.	El problema de la planificación	125
4.4.1.	Ejecutivo cíclico	125
4.4.2.	Algoritmos de planificación de tareas	128
4.4.3.	Criterios de planificabilidad de tareas	129
4.5.	Modelo general de tareas	133
4.6.	Inversión de prioridad	133
4.6.1.	Sección crítica no expulsable	135

4.6.2.	Herencia de prioridad	137
4.6.3.	Techo de prioridad	139
4.7.	Planificación de tareas aperiódicas	140
4.7.1.	Servicio de procesamiento en segundo plano	140
4.7.2.	Servicio de peticiones utilizando una tarea sondeante	141
4.7.3.	Servicio de peticiones utilizando un servidor diferido	142
5.	Relaciones de problemas	145
5.1.	Introducción	145
5.2.	Sincronización en Memoria Compartida	194
5.2.1.	Exclusión mutua	194
5.2.2.	Monitores	211
5.3.	Paso de mensajes	241
5.4.	Sistemas de Tiempo Real	258

1. Introducción a la Programación Concurrente

1.1. Conceptos básicos

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

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

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

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

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

1.1.1. Comparación de programas concurrentes con secuenciales

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

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

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

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

1.1.2. Definición de concurrencia

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

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

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

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

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

- Da herramientas, instrucciones y sentencias útiles para problemas de sincronización entre procesos.
- Las primitivas de programación en un lenguaje de alto nivel (como son los lenguajes concurrentes) son más fáciles de utilizar que con lenguajes de bajo nivel. Por ejemplo, resolver un problema de sincronización mediante semáforos es más compleja que la resolución mediante 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 repercute 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.
- Todo proceso debe seguir progresando durante la ejecución de un programa.
- Progreso local: Una vez que un proceso comienza a ejecutar una sección de código, debe terminar dicha sección.
- Progreso global: Si un proceso está listo para entrar en ejecución, el computador debe permitir su ejecución.

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

- El estado de la memoria asignado a cada proceso.
- El valor de los registros de cada proceso.

- El coste computacional de los cambios de contexto.
- La política de planificación que se emplea de los procesos.
- El desarrollo de los programas es independiente del hardware.

1.2. Modelos para creación de procesos en un programa

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

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

1.2.1. Grafos de Sincronización

Un Grafo de Sincronización es un Grafo Dirigido Acíclico (DAG) donde cada nodo representa una secuencia de sentencias del programa (o actividad). Nos sirven para definir situaciones de 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 .

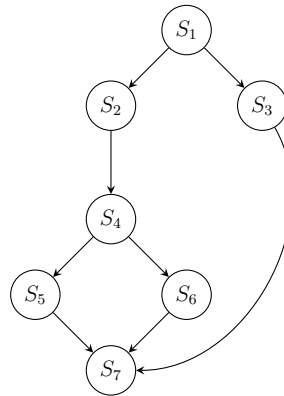


Figura 1.1: Ejemplo de DAG de sincronización.

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 cualquiera de los dos siguientes:

```

1 cobegin
  P1;
  P2;
coend

```

```

1 cobegin P1 || P2 coend

```

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

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

```

1 begin
  S1;
  cobegin
    begin
      S2;
      S4;
      cobegin
        S5;
        S6;
      coend
    end
  end
  S3;

```

```
15  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:

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

1.3. Exclusión mutua y sincronización

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

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

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

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

Ejemplo. Por ejemplo, ante el siguiente código concurrente:

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

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

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

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

Sin embargo, ante el siguiente código concurrente:

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

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

Paradigma del Productor Consumidor

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

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

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

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

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

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

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

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

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

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 progreso finito expuesto en secciones anteriores no tiene nada que ver con la ausencia de inanición:

- El axioma de progreso 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 progreso 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. Además, no pueden arrebatarse 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 interbloques, 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 programas concurrentes, notemos primero que muchos de ellos están pensados 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 estados¹ concretos que un programa alcanza.

De esta forma, el único enfoque posible es el razonamiento asertivo, por ser el que mejor tolera los *errores transitorios* (aquellos que aparacen en unas ejecuciones pero no en otras, lo que dificulta su detección).

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 formadas².
- 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:

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

De forma que disponemos de n hipótesis (H_1, H_2, \dots, 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:

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

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

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 $\{p, q, r, \dots\}$.
- Operadores lógicos $\{\neg, \wedge, \vee, \rightarrow, \leftarrow, \longleftrightarrow\}$.
- Expresiones formadas por constantes, variables y operadores.

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

De esta forma, diremos que:

- 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.

³Cuyos elementos interpretamos como verdadero y falso.

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 precondition 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 letras 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 $\{V\} \text{ while true do begin end } \{F\}$ 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:

- $\{V \wedge P\} \equiv \{P\}$.

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

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 $\{P\} \text{ null } \{P\}$.

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 fuerte la precondition 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 precondition 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*.

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

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

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

es cierta, tendremos que probar que las tripletas

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

son ciertas.

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

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

Donde a la proposición I la llamaremos invariante de dicho bucle.

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 precondition 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:

$$\begin{array}{c} \{x = 0\} \ x = x + 2 \ \{x = 2\} \\ \{V\} \ x = 0 \ \{x = 0\} \end{array}$$

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 $\langle x = x + 2; \rangle$ (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 $\langle x = 0; \rangle$ (en caso de que esta se ejecute después).
- Notemos que la ejecución de una instrucción no *interfiere* con la precondition 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:

$$\begin{aligned} \{x = 0\} \quad x = x + 2 \quad \{x = 0 \vee x = 2\} \\ \{V\} \quad x = 0 \quad \{x = 0 \vee x = 2\} \end{aligned}$$

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 atómicas, 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 corresponde 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 modificada mientras la expresión resulta evaluada.

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

```
1  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 sirva 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 atómica⁴ a con precondition $pre(a)$, 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 $i \neq j$.

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).

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

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.

$$\frac{\{P_i\} \ S_i \ \{Q_i\} \text{ son triples libres de interferencia } 1 \leq i \leq n}{\{P_1 \wedge P_2 \wedge \dots \wedge P_n\} \ \text{cobegin } S_1 \parallel S_2 \parallel \dots \parallel S_n \ \text{coend } \{Q_1 \wedge Q_2 \wedge \dots \wedge Q_n\}}$$

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

$$\begin{array}{c} \{x = 0\} \\ \text{cobegin } x = x + 2; \parallel x = 0; \ \text{coend} \\ \{x = 0 \vee x = 2\} \end{array}$$

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 leída 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 programas 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

$$m \leq n$$

Así mismo, el productor no puede escribir en x más veces de las que el consumidor haya leído 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 leído la variable. De esta forma, limitamos al productor de generar más datos de las que el productor pueda consumir, con la expresión:

$$n \leq m + 1$$

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

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

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 leído todavía x , por lo que:

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

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 $m \leq n \leq m + 1$, debemos probar que tras la escritura de x por el productor y tras la lectura de x por el consumidor se verifica $m' \leq n' \leq m' + 1$, siendo m' y n' el número actualizado de veces que se ha leído y escrito en la variable, respectivamente.

Para ello, suponemos que se verifica I . Como demostramos anteriormente la corrección parcial de cada 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 leído (o que el consumidor habrá leído un dato y el productor habrá generado el siguiente). En cualquier traza de ejecución, tendremos que:

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

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

$$m \leq n \leq m + 1 \implies m + 1 \leq n + 1 \leq m + 2 \implies m' \leq n' \leq m' + 1$$

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 necesario 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
    {Cuerpo del bucle}
end
```

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

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:

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

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 $n+1$ iteraciones, 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.

<pre> 1 var turno : integer; Process P1(); begin 5 while true do begin { Acceso a la SC } while turno <> 1 do begin 10 null; end do { Sección crítica } turno := 2; end do 15 end </pre>	<pre> 1 Process P2(); begin 5 while true do begin { Acceso a la SC } while turno <> 2 do begin 10 null; end do { Sección crítica } turno := 1; end do 15 end </pre>
--	--

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

<pre> 1 var c1, c2 : integer; Process P1(); begin 5 c1 := 1; while true do begin 10 { Acceso a la SC } { Si P2 entró } while c2 = 0 do begin null; end do 15 { Entra a la sección crítica } c1 := 0; { Sección crítica } c1 := 1; 20 end do end </pre>	<pre> 1 Process P2(); begin 5 c2 := 1; while true do begin 10 { Acceso a la SC } { Si P1 entró } while c1 = 0 do begin null; end do 15 { Entra a la sección crítica } c2 := 0; { Sección crítica } c2 := 1; 20 end do end </pre>
--	---

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 da una situación de interbloqueo, que incumpliría la cuarta condición de Dijkstra, al no poder alcanzar nunca la sección crítica (además de la segunda, ya que la bondad de la solución depende de la velocidad de ejecución de los procesos).

<pre> 1 var c1, c2 : integer; Process P1(); begin 5 c1 := 1; while true do begin { Acceso a la SC } 10 c1 := 0; { Si P2 entró } while c2 = 0 do begin 15 null; end do { Sección crítica } c1 := 1; end do end end </pre>	<pre> 1 Process P2(); begin 5 c2 := 1; while true do begin { Acceso a la SC } 10 c2 := 0; { Si P1 entró } while c1 = 0 do begin 15 null; end do { Sección crítica } c2 := 1; end do end end </pre>
---	--

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

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 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.

```
1  var c1, c2 : integer;

Process P1();
begin
5   c1 := 1;

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

```
1  Process P2();
begin
5   c2 := 1;

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

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

- 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.

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.

```

1  var c1, c2, turno : integer;

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

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

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

20                           end
                               end do

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

```

```

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

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

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

20                           end
                               end do

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

```

Figura 2.5: Algoritmo 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 progra-

mas 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 encuentra 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.

```

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

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

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

            c[i] := en_SC;

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

            { Sección crítica }
            c[i] := pasivo;
30        end do
    end
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 suceder que dos (o más, en cuyo caso es una explicación análoga) procesos, P_i y P_j (con $i, j \notin \{\text{turno}\}$) 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 Dijkstra:

Exclusión mutua.

La propiedad de exclusión mutua está garantizada gracias al bucle `B`:

El proceso P_i (con $i \in \{0, \dots, n-1\}$) 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, \dots, 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 $k \in \{1, \dots, m\}$, 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 $\text{turno} = 0$ 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 $\text{turno} = 2$) y luego P_1 (cambia $\text{turno} = 1$), 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 da la propiedad de equidad de la solución.

Donald Knuth se dio 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.

```

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

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

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

20             c[i] := en_SC;

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

30             turno := i;
                { Sección crítica }
                turno := (i-1) mod n;
                c[i] := pasivo;
            end do
35 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 $\text{turno} = i$, entonces P_i se salta el bucle A. Si no ($\text{turno} \neq i$), 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, \dots, P_m que tratan de acceder a la sección crítica a la vez, y suponiendo que **turno** = k :

- Si $k \in \{1, \dots, m\}$, 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 $k \notin \{1, \dots, m\}$, entonces tenemos que $c[k] = \text{pasivo}$. 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 $c[j] \neq \text{pasivo}$ 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 demostrar 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 $victima \in \{1, \dots, n\}$) 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 a la sección crítica¹, ya que si no (esto es, $P_{victima}$ es el siguiente en turno rotatorio tras P_i) tras salir P_i de la sección crítica ejecutaría la sentencia $\text{turno} := (i-1) \bmod n$, 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 k pero anterior a $victima$ que también adelanta 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 propiedad 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 $f : \mathbb{N} \rightarrow \mathbb{N}$ una función que contabiliza el número máximo de adelantamientos, $f(n)$ para un escenario con n procesos, entonces:

¹Notemos 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 **solicitando** y como $c[1] = \text{solicitando}$, 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 sección 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, \dots, P_{n-1} de forma que inicialmente $\text{turno} = n - 2$ 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 $j = n - 3$, 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 $\text{turno} = n - 1$ y P_{n-1} pueda ejecutar la sección crítica queremos alejarlo tanto como podamos. Para ello, buscamos el peor momento para tener $\text{turno} = n - 2$, 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 $\text{turno} = n - 3$, 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:

$$f(n) = f(n-1) + 1 + f(n-1) = 2f(n-1) + 1 \quad \forall n \geq 2$$

Recurrencia que podemos resolver², obteniendo que:

$$f(n) = 2^{n-1} - 1 \quad \forall n \geq 1$$

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;
```

²Tal 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.

<pre> 1 Process P1(); begin while true do begin 5 { Acceso a la SC } c[1] := true; turno := 1; while (c[2] and turno = 1) do 10 begin null; end do { Sección crítica } 15 c[1] := false; end do end end </pre>	<pre> 1 Process P2(); begin while true do begin 5 { Acceso a la SC } c[2] := true; turno := 2; while (c[1] and turno = 2) do 10 begin null; end do { Sección crítica } 15 c[2] := false; end do end end </pre>
--	--

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, $c[j] = \text{true}$, 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.

Supuesto que dos procesos intentan 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, contradiciendo 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 llamaremos *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:

```
1 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 $k \in \{0, \dots, n - 1\}$, 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:

```
1 Process Pi();
  begin
    while true do
      begin
5       { - Acceso a la SC - }
        for j=0 to n-2 do
          begin
            c[i] := j;
            turno[j] := i;
10
            { Existe k <> i con c[k] >= j AND turno[j] = i }
            for k=0 to n-1 do
              begin
                if (k = i) then continue;
15                while (c[k] >= j and turno[j] = i ) do
                  null;
                end do
              end
            end
          end
20
        c[i] := n-1;      { Proceso en etapa n-1 (SC) }
```

```

    { - Sección Crítica - }
    c[i] := -1;
end do
25 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:

```

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

```

Es decir, si $\text{turno}[j] = i$ (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 $c[k] \geq j$, 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 ($\text{turno}[j] \neq i$).
- No haya ningún proceso en ninguna etapa siguiente (es decir, que todos los procesos que antes habían pasado al nuestro han terminado ya la ejecución de la sección crítica, con lo que no existe un k de forma que $c[k] \geq j$).

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:

```

1 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	$c[0]$	$c[1]$	$c[2]$	$c[3]$	$\text{turno}[0]$	$\text{turno}[1]$	$\text{turno}[2]$
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: Evolución 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 $c[k] \geq 1$), 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_0 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 una etapa posterior a la etapa de P_k .

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

$$\text{Exists } k \neq i : c[k] \geq j \text{ AND } \text{turno}[j] = i.$$

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 $j + 1$. \square

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 $c[k] \geq j$, es decir, que $c[k] < j = c[i]$ para todo $k \neq i$, con lo que P_i precede a todos los demás procesos.
- $\text{turno}[j] \neq i$, con lo que existirá otro proceso P_k de forma que haya sido el último en modificar el valor de $\text{turno}[j]$, con lo que dicho proceso P_k ha de estar necesariamente en la etapa j , al igual que P_i .

\square

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 cuando 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 $j - 1$, veámoslo 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 $j - 1$. \square

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, \dots , $j - 1$ (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 $n - j$. \square

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 y calculamos el número máximo de turnos que el proceso ha de esperar para poder entrar en la sección crítica:

El caso más desfavorable posible se da cuando los n procesos intentan acceder a la vez 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. Estos tardarán un turno en propagarse por cada etapa y como tenemos $n - 2$ etapas restantes (más el turno de la etapa 0), llevamos ya $n - 1$ turnos de espera para P_i . Este 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.

En este caso, P_i podrá entrar a la sección crítica, sin esperar un turno más.

- 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.

En este caso, P_i tendrá que esperar tantos turnos como antes para llegar a la sección crítica menos uno, ya que ha avanzado de etapa.

En cualquier caso, para llegar a la sección crítica, P_i debe esperar a que el proceso que se bloqueó en la etapa 1 llegue a 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 que el proceso que se bloquea en la etapa 1 entre a la sección crítica, que resulta en un problema similar pero en este caso de $r(n - 1)$ turnos de espera. De esta forma:

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

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 prácticas³. Los semáforos presentan dos grandes limitaciones:

1. Están basados en variables compartidas del programa, por lo que no fomentan la modularidad de los programas, impidiendo su reutilización.
2. Las operaciones de los semáforos (`sem_wait` y `sem_signal`) se encuentran dispersas a lo largo del código del programa concurrente. Además, estas instrucciones no solo afectan al bloque de código en el que se encuentran, sino a cualquier otro bloque que use el mismo semáforo.

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

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

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.

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

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

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

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

Los procesos que usemos en los programas 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 invariantes 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 procedimientos⁵ 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*

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

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 engloba:

- 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 adelante⁶.

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

```

⁶Como el tipo de dato cond.


```

1  Proceso Cons1::
    var x : tipo_dato;

    while true do begin
5     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):

```

1  Monitor Buf
    var
        -elementos_ocupados : int;
        -frente, atras: 0..N-1;
5     -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):

```

1  procedure insertar(d : tipo_dato) begin
    if((frente + 1) mod N = atras) then no_lleno.wait();
    introducir(buf, frente, d); {inserta d en la posición frente en el buffer}
    elementos_ocupados += 1;
5     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):

```

1  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;
5     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 $(\text{frente} + 1) \bmod N = \text{atras}$, 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 buffer⁷.

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 dispone 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 proceso⁸.

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

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

⁸En 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.

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 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 proceso⁹). 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 localmente¹⁰ 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

⁹Se explicará más adelante.

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

los monitores inicializan sus variables permanentes¹¹.

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

```

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

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

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

    begin
15 x = inicio; inc = incremento;
    end

```

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

```

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

¹¹Además, 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.

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:

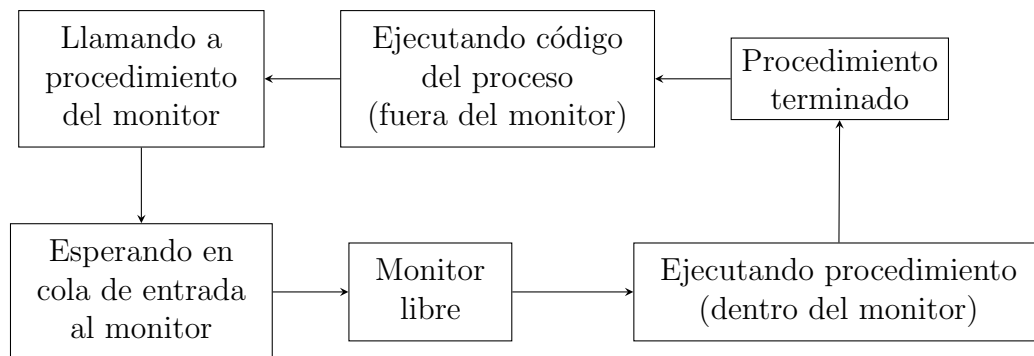


Figura 2.9: Vida de un proceso en un programa concurrente con monitores.

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 desbloquea¹². Si esta cola está vacía, es equivalente a una operación nula¹³.

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

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

¹³Esto es, equivalente a la instrucción ;.

entre los procesos para entrar al monitor, incumpliendo la propiedad de equidad entre procesos. Depende de la semántica de las señales del lenguaje¹⁴. Se recomienda **no usarla**.

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 monitor¹⁵, 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

¹⁴Se explicará más adelante qué es esto.

¹⁵Veremos más soluciones.

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.

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 precondition o poscondition del proceso bloqueado, por lo que tras desbloquearse, no podemos esperar nada de dicho proceso.

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

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

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 cumpliendo, 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. Finalmente, 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 IM¹⁷, 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.

¹⁷A 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):

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

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

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

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

Sin embargo, en caso de ejecutar el siguiente código:

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

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

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 después¹⁹ de dicha instrucción `wait`.

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

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

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**:

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

o equivalentemente:

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

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

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

¹⁹Gracias a que estamos en lenguajes reentrantes.

Observación. Notemos que el axioma de la operación **signal** funciona porque hemos supuesto 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

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

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

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

Como el triple $\{V\} \ s = 0; \ \{s = 0\}$ es cierto por el axioma de asignación y tenemos que $\{s = 0\} \rightarrow \{s \geq 0\}$, 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

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

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

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

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

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

Por tanto, basta probar el triple $\{s > 0\} \ \text{null}; \ \{s > 0\}$, 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 precondition del **signal** asociado a la misma variable condición, que se encuentra en el procedimiento V. Para hallar la precondition de la instrucción **c.signal**, tendremos que demostrar alguna instrucción de dicho procedimiento, con el fin de hallar la precondition.

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

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

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:

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

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

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

- 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

$$\begin{aligned} &\{IM\} \ s = s + 1; \ \{s > 0\} \\ &\{s > 0\} \ c.signal; \ \{IM\} \end{aligned}$$

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:

- $np \leq na$, ya que para completar una llamada P hay que realizar una llamada.
- $np \leq nv$, ya que, como inicialmente $np = na = nv = 0$, para poder completar una llamada a P, hay que previamente haber hecho una llamada a V.
- $np \geq \min(na, nv)$, para no detener innecesariamente a los procesos, cumpliendo la hipótesis de progreso finito.

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

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

Pasemos ahora a la demostración del monitor:

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

$$\begin{aligned}
 & \{V\} \\
 & na = 0; np = 0; nv = 0; \\
 & \{na = 0 \wedge np = 0 \wedge nv = 0\} \rightarrow \{IM\}
 \end{aligned}$$

2. Posteriormente, probaremos el triple del procedimiento P:

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

para ello:

$$\begin{aligned}
 & \{IM\} \equiv \{np = \min(na, nv)\} \\
 & na = na + 1; \\
 & \{np = \min(na - 1, nv)\} \\
 & \text{if } (na > nv) \text{ then} \\
 & \{na > nv \wedge np = \min(na - 1, nv)\} \rightarrow \{na - 1 \geq nv \wedge np = \min(na - 1, nv)\} \rightarrow \\
 & \rightarrow \{na > nv \wedge np = nv\} \rightarrow \{np = \min(na, nv)\} \\
 & c.wait();
 \end{aligned}$$

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

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

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

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

$$\begin{aligned}
\{IM\} &\equiv \{np = \text{mín}(na, nv)\} \\
&\quad na = na + 1; \\
&\quad \{np = \text{mín}(na - 1, nv)\} \\
&\quad \text{if } (na > nv) \text{ then} \\
&\quad \{na > nv \wedge np = \text{mín}(na - 1, nv)\} \rightarrow \{na - 1 \geq nv \wedge np = \text{mín}(na - 1, nv)\} \rightarrow \\
&\quad \rightarrow \{np = nv\} \rightarrow \{np = \text{mín}(na, nv)\} \\
&\quad \quad c.wait(); \\
&\quad \{na > np \wedge np = nv - 1\} \\
&\quad \text{else do} \\
&\quad \{na \leq nv \wedge np = \text{mín}(na - 1, nv)\} \rightarrow \{na - 1 < nv \wedge np = \text{mín}(na - 1, nv)\} \rightarrow \\
&\quad \rightarrow \{na - 1 < nv \wedge np = na - 1\} \rightarrow \{np < nv \wedge np = na - 1\} \\
&\quad \quad \text{null;} \\
&\quad \{np < nv \wedge np = na - 1\} \\
&\quad \text{endif}
\end{aligned}$$

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:

$$\begin{aligned}
\{na > np \wedge np = nv - 1\} &\rightarrow \{na > nv - 1 \wedge np = nv - 1\} \rightarrow \\
&\rightarrow \{na \geq nv \wedge np = nv - 1\} \\
\{np < nv \wedge np = na - 1\} &\rightarrow \{na - 1 < nv \wedge np = na - 1\} \rightarrow \\
&\rightarrow \{na \leq nv \wedge np = na - 1\}
\end{aligned}$$

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:

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

3. Para probar ahora el triple del procedimiento V:

$$\begin{aligned}
\{IM\} &\equiv \{np = \text{mín}(na, nv)\} \\
&\quad nv = nv + 1; \\
&\quad \{np = \text{mín}(na, nv - 1)\} \\
&\quad \text{if } (na > np) \text{ then} \\
&\quad \quad \{na > np \wedge np = \text{mín}(na, nv - 1)\} \rightarrow \{na > np \wedge np = nv - 1\} \\
&\quad \quad c.signal(); \\
&\quad \quad \{np = \text{mín}(na, nv)\} \\
&\quad \text{else do} \\
&\quad \quad \{na \leq np \wedge np = \text{mín}(na, nv - 1)\} \rightarrow \{np = na\} \rightarrow \{np = \text{mín}(na, nv)\} \\
&\quad \quad \text{null}; \\
&\quad \quad \{np = \text{mín}(na, nv)\} \\
&\quad \text{endif} \\
&\quad \{np = \text{mín}(na, nv)\} \equiv \{IM\}
\end{aligned}$$

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 $i \in \{1, \dots, n\}$ 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 $1 \leq k \leq m$, entonces podemos aplicar la **regla de concurrencia para programas con monitores**:

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

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 procedimientos 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:

```

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

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

   c : cond;

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

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

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

```

Para su demostración, usaremos el invariante

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

Demostración. Hemos pues de demostrar que se cumplen los triples:

$$\begin{aligned} &\{V\} \text{ codigo inicializacion } \{IM\} \\ &\{IM\} \text{ esperar } \{IM\} \\ &\{IM\} \text{ notificar } \{IM\} \end{aligned}$$

Comenzamos por el primero:

$$\{V\} \text{ terminado} = false; \text{ autorizado} = false; \{IM\}$$

Para ello, hemos de aplicar la regla de la composición sobre los triples:

$$\{V\} \text{ terminado} = false; \{terminado = false\}$$

$$\begin{aligned} &\{terminado = false\} \text{ autorizado} = false; \\ &\{terminado = false \wedge autorizado = false\} \rightarrow \{IM\} \end{aligned}$$

Que son ambos ciertos, el primero por el axioma de asignación y el segundo por el axioma de asignación y la primera regla de la consecuencia.

A continuación, probaremos los triples de los procedimientos del monitor, recordando que para hallar la poscondición de `c.wait()` hemos primero de localizar la precondición de `c.signal()`, por lo que hemos de realizar las dos demostraciones

“al mismo tiempo”, aunque a continuación escribiremos primero la de `notificar` y luego la de `esperar`:

```

notificar(); begin
    {IM}
    terminado = true;
    {terminado = true}  $\equiv$  {C}
    c.signal();
    {IM}
end

```

Con lo que ya sabemos que la condición de sincronización de `c` es:

$$\{C\} \equiv \{\text{terminado} = \text{true}\}$$

Demostramos ahora $\{IM\}$ `esperar` $\{IM\}$:

```

esperar(); begin
    {IM}
    if not terminado then
        {IM  $\wedge$  terminado = false}  $\rightarrow$  {terminado = false  $\wedge$  autorizado = false}
        c.wait();
        {terminado = true}
    else
        {IM  $\wedge$  terminado = true}  $\rightarrow$  {terminado = true}
        null;
        {terminado = true}
    end
    {terminado = true}
    autorizado = true;
    {terminado = true  $\wedge$  autorizado = true}  $\rightarrow$ 
     $\rightarrow$  {autorizado = true  $\implies$  terminado = true}  $\equiv$  {IM}
end

```

□

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 crítica²⁰.

²⁰Podemos 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.

Usualmente, queremos 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:

```

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

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

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

15 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

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

Demostración. Tenemos que demostrar:

$$\begin{aligned}
 &\{V\} \text{ codigo inicializacion } \{IM\} \\
 &\{IM\} \text{ entrar } \{IM\} \\
 &\{IM\} \text{ salir } \{IM\}
 \end{aligned}$$

Comenzamos por el código de inicialización:

$$\{V\} \text{ ocupada} = \text{false}; \{ocupada = false\} \rightarrow \{IM\}$$

Ahora, comprobamos el triple del procedimiento **salir**:

```

    salir(); begin
        {IM}
        ocupada = false;
        {IM ∧ ocupada = false}
        cola.signal();
        {IM}
    end

```

Con lo que ya sabemos que la condición de sincronización de la variable condición c es $\{IM \wedge ocupada = false\}$. Finalmente, el triple del procedimiento **entrar**:

```

    entrar(); begin
        {IM}
        if ocupada then
            {IM ∧ ocupada = true}
            cola.wait();
            {IM ∧ ocupada = false}
        else
            {IM ∧ ocupada = false}
            null;
            {IM ∧ ocupada = false}
        end
        {IM ∧ ocupada = false}
        ocupada = true;
        {IM ∧ ocupada = true} → {IM}
    end

```

□

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`:

```

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

5

   procedure escribir(v : integer); begin
      if pendiente then
         cola_prod.wait();
         valor = v;
10      pendiente = true;
         cola_cons.signal();
      end

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

   begin
      pendiente = false;
   end

```

Para la demostración, debemos definir primero:

E = número de llamadas a escribir **completadas**.

L = número de llamadas a leer **completadas**.

De esta forma, podemos definir el invariante

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

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 demostrar 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 momento²¹. 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 encontrarlos, 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.

²¹Evitando que entre al monitor cualquier otro proceso de la cola de entrada al monitor.

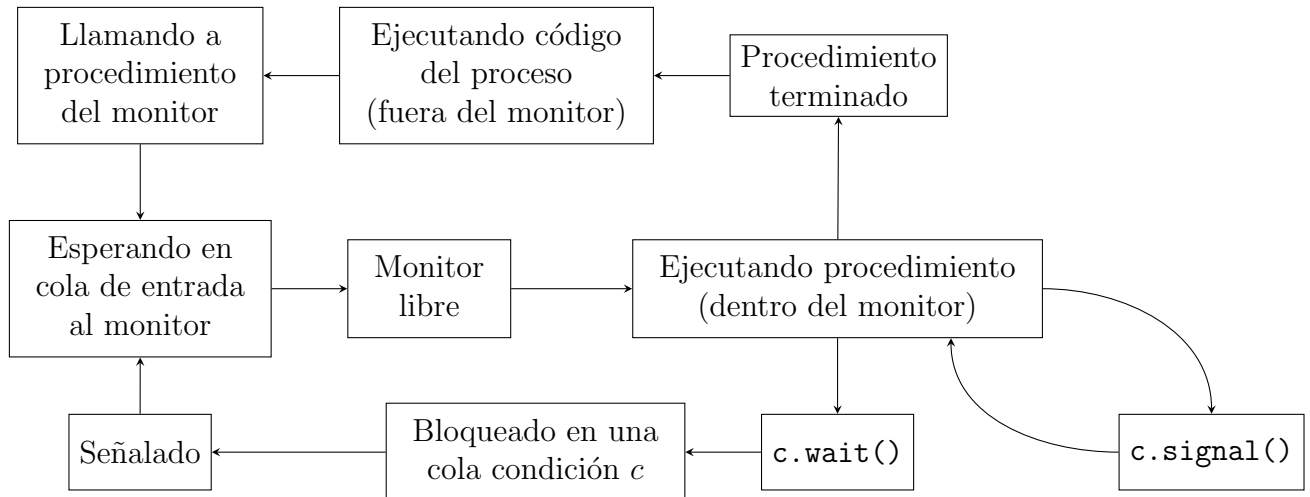


Figura 2.10: Vida de un proceso en un programa con monitores con semántica SC.

- 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.

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 de 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

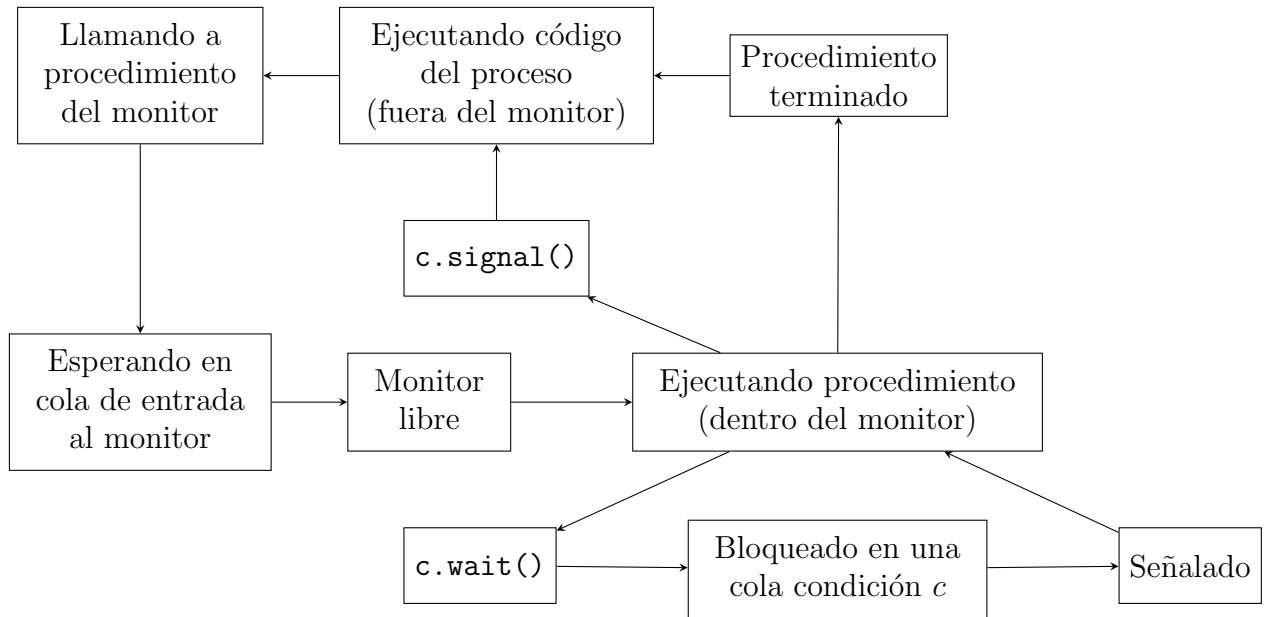


Figura 2.11: Vida de un proceso en un programa con monitores con semántica SS.

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.

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 procesos 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.

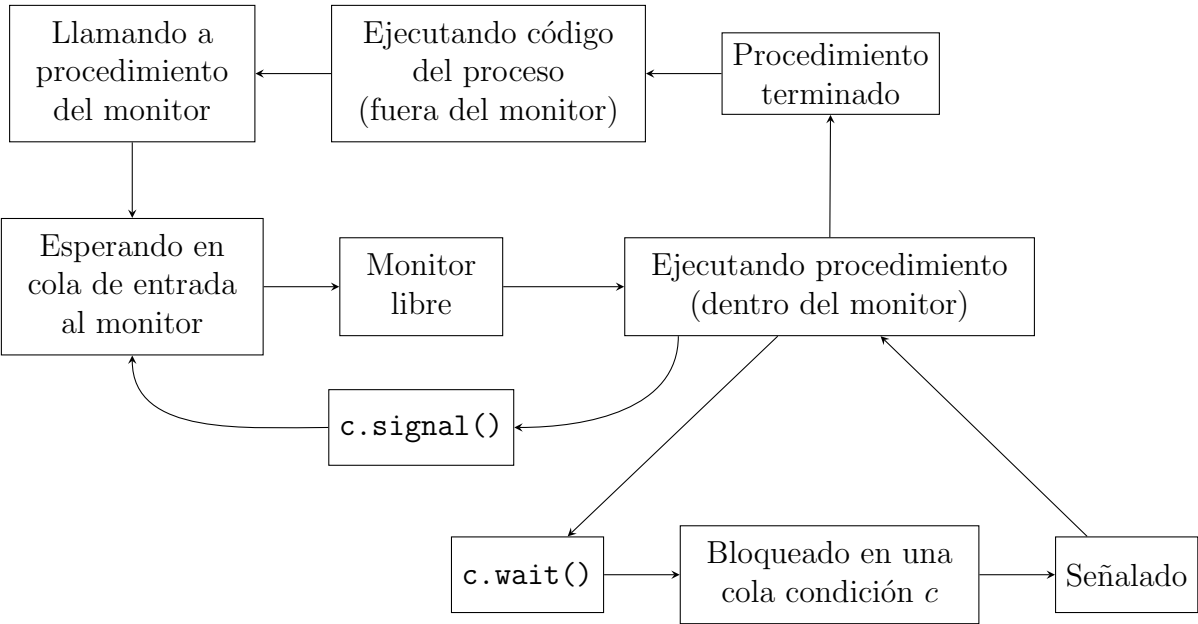


Figura 2.12: Vida de un proceso en un programa con monitores con semántica SE.

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 usa monitores con semántica de señalar y espera urgente con el diagrama de la Figura 2.13.

Finalmente, recordemos que con semántica SU, si la cola de una variable condición está vacía y realizamos un `signal` sobre la misma, será equivalente a una operación nula, con lo que en este caso el proceso señalador no se irá a la cola de bloqueados.

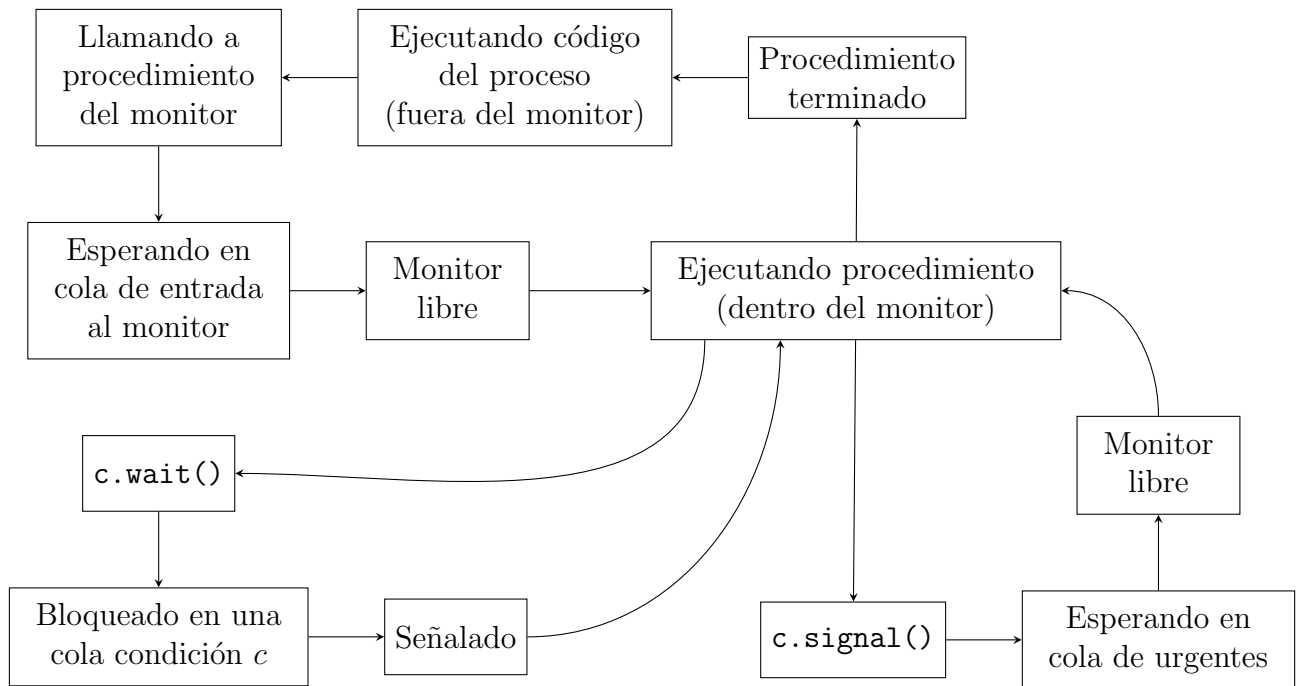


Figura 2.13: Vida de un proceso en un programa con monitores con semántica SU.

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 momento 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 cuanto 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 de señales más eficiente.

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 Computadores²², 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 le 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 volver 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

```

²²Consultar los apuntes si no se conoce el concepto.

```

20  contador = 0;
    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 finalice 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
15  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, \dots , 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:

```

1  Monitor semaforo_FIF01;
   var c : cond;
       s : int;

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

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

   begin
       s := 0;
20  end

```

```

1  Monitor semaforo_FIF02;
   var c : cond;
       s : int;

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

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

   begin
20  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, observamos 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 monitor²³).
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.

²³Notemos que esto no incumple 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, (donde P4 y P2 tienen la misma prioridad, pero como P4 se bloquea después, decidimos colocarlo en la cola después de P2) 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 urgentes²⁴, 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.

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

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

```
1 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:

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

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

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

Y el de salida:

```
1 procedure salida();
  begin
    if(next_count <> 0)    {hay procesos en cola de urgentes}
      sem_signal(next);
5    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**):

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

```
5      if(next_count <> 0) then  {hay procesos en cola de urgentes}
        sem_signal(next);
      else
        sem_signal(mutex);  {deja libre el monitor}
      end if
10
      sem_wait(x_sem);
      x_count := x_count - 1;
    end
```

```
1  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;
5      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 sobreuso 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 paralela 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 datos¹.

¹Algo 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 accedan a una zona de memoria común es muy caro, ya que hace falta una interconexión muy rápida entre esta y todos los multiprocesadores, si no queremos perder la ganancia que hemos ganado al pasar de un sistema uniprocador 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, bastará 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 usaremos² 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 modificadas 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).

²También 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 diferente, 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** cuándo la ejecución de esta garantiza que el valor recibido por el destinatario sea el valor que tenían los datos antes de la llamada.

Por tanto, 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.

Normalmente, aquellas implementaciones que usen alguna instrucción (**send** o **receive**, o ambas) insegura irán acompañadas de instrucciones capaces de determinar cuando es posible utilizar (modificar en el caso de **send** y consultar en el caso de **receive**) los datos involucrados con dicha operación insegura, con el objetivo de utilizar estos datos de forma segura³.

La razón de existencia de las instrucciones de comunicación inseguras es priorizar eficiencia, con la contrapartida de que el programador debe ser cuidadoso a la hora de usar dichas instrucciones.

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 el 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. En la siguiente tabla se muestran resumidos, y los desarrollaremos a continuación:

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

³La utilización de instrucciones inseguras puede llegar a ser tan seguro como la utilización de instrucciones seguras, con la diferencia de que en las primeras es responsabilidad del programador el buen uso de las instrucciones.

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:

```

1 Process P0;
  var x0;
  begin
    send(&dato1, P1);
5    receive(&x0, P1);
  end

```

```

1 Process P1;
  var x1;
  begin
    send(&dato2, P0);
5    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 sino 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 que tan solo debe esperar a la realización 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 interrupción 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 consecuencia, se reduce el tiempo de espera en el proceso receptor durante el cual un acceso a dichos datos es inseguro.

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 comportamiento. 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:

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.

<pre>1 Process P1; begin while true do begin while(no_pasa_persona()) do begin 5 null; end send(avisos, P3); end end end</pre>	<pre>1 Process P2; begin while true do begin while(no_pasa_persona()) do begin 5 null; end send(avisos, P3); end end end</pre>
--	--

Figura 3.2: Código para las puertas del museo.

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.

```
1  P3;  
   var aforo : integer;  
   begin  
     while true do begin  
5      receive(avisos, P1);  
       aforo := aforo + 1;  
       receive(avisos, P2);  
       aforo := aforo + 1;  
10      { Resto de sentencias }  
     end  
   end
```

Figura 3.3: Primer código para el controlador.

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 guarda⁴, 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 orden 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. 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 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.

⁴Quién lo habría dicho.

```

1  P3;
   var aforo : integer;
   begin
   while true do begin
5     if
       (receive(avisos, P1))
       []
       (receive(avisos, P2))
     fi
10    aforo := aforo + 1;
       { Resto de sentencias }
   end
   end

```

Figura 3.4: Código para el controlador usando órdenes con guarda.

3.4.1. Definición de sentencia de espera selectiva o select

Una sentencia de espera selectiva 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 y una instrucción **receive**.
De esta forma, si la condición booleana es evaluada como **false**, la instrucción **receive** no llegará a ejecutarse.
- Que sea una condición booleana.
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:

```

1  select
   when condicion1 receive(variable1, proceso1) do
       sentencias;
   when condicion2 receive(variable2, proceso2) do
5     sentencias;

```

```
...  
when condicionn receive(variablenn, 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 clasificarlas 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 cierta⁵. En caso de no disponer de condición booleana pero sí de instrucción `receive`, 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, que 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:

⁵Podemos pensar que el razonamiento anterior es cierto por vacuidad.

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:

- 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 proceso 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.

```

1  {Identificador: P}
   Process Productor;
   var v : integer;
   begin
5  while true do begin
       v := Producir();
       send(v, B);
   end
end

```

```

1  {Identificador: C}
   Process Consumidor;
   var v : integer;
   begin
5  while true do begin
       send(s, B);
       receive(v, B);
       Consumir(v);
   end
10 end

```

Figura 3.5: Código para el problema de los productores/consumidores.

Así, el productor 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.

```

1  {Identificador: B}
   Process Intermedio;
   var esc, lec, ocupados : integer := 0;
       buf : array[0..tam-1] of integer;
5  begin
   while true do begin
       select
           when ocupados < tam receive(v, P) do
               buf[esc] := v;
10          esc := (esc + 1) mod tam;
               ocupados := ocupados + 1;
           when 0 < ocupados receive(s, C) do
               send(buf[lec], C);
15          lec := (lec + 1) mod tam;
               ocupados := ocupados - 1;
       end
   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 solamente 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* + 1 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.

```

1  select
    for i := 0 to n-1
        when suma[i] < 100 receive(numero, fuente[i]) do
            suma[i] := suma[i] + numero;
5  end

```

Figura 3.7: Código con guardas indexadas.

Este código será equivalente al de la Figura 3.8

```

1  select
    when suma[0] < 100 receive(numero, fuente[0]) do
        suma[0] := suma[0] + numero;
    when suma[1] < 100 receive(numero, fuente[1]) do
5  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 proceso 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
       end
15  end
```

Figura 3.9: Código del proceso servidor.

4. Sistemas de Tiempo Real

Un Sistema de Tiempo Real (STR) es aquel en el que la respuesta correcta a un cálculo realizado por el programa no sólo depende de que efectivamente haga lo que tenga que hacer, sino también de cuándo esté disponible dicho resultado. A lo largo de este Capítulo, usaremos el término “tarea” para referirnos a un proceso que interviene en nuestro sistema.

En el ámbito de los sistemas operativos, el estándar POSIX define a un sistema operativo de tiempo real como aquel que tiene la capacidad para suministrar un nivel de servicio requerido en un tiempo limitado y especificado de antemano. Es decir un sistema de este tipo debe permitir satisfacer plazos de entrega prefijados a todos sus procesos que estén etiquetados como de tiempo real.

Hay una serie de elementos o propiedades característicos que poseen todos los sistemas de tiempo real y que nos pueden ayudar a identificarlos correctamente:

- **Reactividad:** se dice que los STR son sistemas reactivos porque su funcionamiento se basa en una interacción continua con su entorno, a diferencia de los transformacionales cuyo comportamiento abstracto es parecido al de una función matemática: entrada de datos, cálculos y salida de resultados.
- **Determinismo:** es una cualidad clave en los sistemas de tiempo real. Es la capacidad de determinar con una alta probabilidad, cuánto es el tiempo que tarda una tarea en iniciarse. Esto es importante porque los sistemas de tiempo real necesitan que ciertas tareas se ejecuten antes de que otras se puedan iniciar. Este dato es importante saberlo porque casi todas las peticiones de interrupción se generan por estímulos que provienen del entorno del sistema, así que resulta muy importante para poder determinar el tiempo que el sistema tardará en dar el servicio.
- **Responsividad:** esta propiedad tiene que ver con el tiempo que tarda una tarea en ejecutarse una vez que la interrupción ha sido atendida. Los aspectos a los que se enfoca son:
 - La cantidad de tiempo que se lleva el iniciar la ejecución de una interrupción.
 - La cantidad de tiempo que se necesita para realizar la tarea que solicitó la interrupción.
 - Los efectos de interrupciones anidadas.
- **Confiabilidad:** es otra característica clave en un sistema de tiempo real. El sistema no debe sólo estar libre de fallas sino, más aún, la calidad del servicio

que presta no debe degradarse más allá de un límite determinado. El sistema debe de seguir en funcionamiento a pesar de catástrofes, o fallas mecánicas. Usualmente una degradación en el servicio en un sistema de tiempo real lleva consecuencias catastróficas.

Clasificación de los STR

Podemos clasificar a los STR en función de su criticidad, distinguiendo entre:

No permisivos o de misión crítica (Hard Real-Time Tasks).

Un sistema de misión crítica es aquel en que es inadmisibile que los resultados lleguen tarde. De esta forma, la pérdida de un tiempo límite supondría un fallo total del sistema.

Un ejemplo de sistema de misión crítica puede ser el software de control de despliegue del tren de aterrizaje de un avión, donde el funcionamiento del sistema fuera de los límites de tiempo establecidos puede llevar a grandes catástrofes.

Permisivos o suaves (Soft Real-Time Tasks).

Un sistema permisivo es aquel en el que el incumplimiento de un plazo en la ejecución de una tarea tiene como mayor repercusión en el sistema la pérdida de rendimiento durante un corto periodo de tiempo que, según su magnitud, puede llegar hasta a ser aceptable en la versión final de un producto.

Si en un sistema de adquisición de datos meteorológicos un componente no es capaz de realizar una determinada funcionalidad a tiempo con la consecuencia de tener un error en una décima a la hora de medir la temperatura en un determinado área, no supone un error grave.

Estrictos (Firm Real-Time Tasks).

Entre los dos tipos de STR anteriores podemos encontrar los sistemas de tiempo real estrictos. Este tipo de sistemas son tolerables a pérdidas infrecuentes del tiempo límite de las tareas de tiempo real, aunque dichas pérdidas puedan llegar a degradar la calidad del servicio del sistema. Para distinguir a los sistemas estrictos de los permisivos, cabe destacar que mientras que en los permisivos la obtención de resultados tardíos siguen siendo útiles en el sistema, en los estrictos esto no sucede.

Un ejemplo de sistema de tiempo real estricto es el sistema que podemos encontrar a cargo de realizar la reserva de vuelos en una compañía.

4.1. Consideraciones sobre el Tiempo

Como vamos a estar interesados a lo largo del Capítulo sobre plazos temporales que han de cumplir las tareas, es momento ahora de determinar de forma concreta qué es el tiempo. En la actualidad, contamos con lenguajes de programación de alto nivel y sistemas operativos que son capaces de proporcionarnos relojes de tiempo real que son capaces de medirnos el tiempo con precisión. Sin embargo, hay que tener en cuenta los distintos tipos de tiempos que podemos encontrarnos:

- Medidas de tiempo absoluto.
- Medidas de tiempo por intervalos o tiempo relativo.

4.1.1. Relojes de tiempo real

En el contexto de los sistemas de tiempo real, un reloj es un módulo compuesto por elementos hardware y software entre los que podemos destacar:

- Un circuito *oscilador*, que generará impulsos eléctricos de forma periódica.
- Un *contador* que acumula los impulsos guardando su valor en una palabra de memoria.
- Un *software* capaz de interpretar dicho contador y pasarlo a unidades con las que estemos cómodos trabajando.

Como características importantes que han de tener estos relojes destacamos:

- Precisión o granularidad: cada cuanto tiempo llega un nuevo impulso que cambia la cuenta acumulada.
- Intervalo: el mayor rango de valores de tiempo que es capaz de medir el reloj antes del desbordamiento del contador.

En relación a esta última característica de los relojes de tiempo real, dentro de una aplicación podemos encontrarnos con:

- Tiempos monótonos: el tiempo de vida de la aplicación es menor que el tiempo que tarda el reloj en desbordarse.
- Tiempos no monótonos: la aplicación tiene un tiempo de vida mayor al tiempo de desbordamiento del reloj, por lo que en algún instante futuro podemos obtener un tiempo menor que en un instante presente.

4.1.2. Temporizadores y retardos

Un temporizador en un ordenador es usualmente una llamada al sistema operativo definida por dos parámetros: el tiempo de arranque (instante en el que se llama a dicha rutina del sistema operativo) y el tiempo de parada, que se representa con una cantidad relativa al tiempo de arranque.

Podemos definir temporizadores de un único disparo o temporizadores periódicos, de forma que cada ciertos intervalos envíen una señal.

Debemos tener en cuenta que si establecemos que un programa espere a que un determinado temporizador le avise dentro de un determinado tiempo x (por ejemplo, como resultado de indicar que el proceso se bloquee durante x segundos: `sleep_for(x)`), dicha cantidad de tiempo no será emulada con total exactitud por parte del sistema operativo, sino que siempre debemos tener en cuenta ciertos retrasos que pueden producirse a la hora de determinar cuándo ha pasado esa determinada cantidad de tiempo x (debido por ejemplo a interrupciones del sistema o a

imprecisiones del reloj de tiempo real).

A esta cantidad de error cometido a la hora de determinar intervalos de tiempo futuros se le llama **deriva**. Este pequeño error cometido en temporizadores de único disparo no es de importancia, pero en temporizadores periódicos sí, ya que si queremos que un programa realice una determinada acción cada segundo y para ello hacemos uso de¹ `sleep_for(1000)` con una deriva de 4ms, entonces tras 100 ciclos de dicho programa, el programa llevará un retardo total de 400ms, retrasándose casi medio segundo en la siguiente ejecución de la función correspondiente. Este retardo recibe el nombre de **deriva acumulativa**, y es algo que siempre intentaremos evitar.

Evitar la deriva acumulativa es tan sencillo como usar la operación `sleep_until(x)` en lugar de `sleep_for(x)`, ya que aunque cometamos ciertas derivas locales en cada bloqueo del proceso, estas no se acumularán, al desbloquear el proceso en tiempos absolutos y no relativos.

4.2. Modelo simple de tareas

Con vistas a analizar de forma comprensible el comportamiento de un conjunto de tareas durante su ejecución dentro de un sistema de tiempo real, es necesario imponer algunas restricciones a su estructura e interacciones con el resto de tareas del programa.

Comenzaremos introduciendo el *modelo simple de tareas*, que introduce ciertas restricciones sobre el tipo de tareas que consideramos, permitiendo comenzar con un estudio inicial de los STR para luego relajar este modelo. Antes de introducir las características del modelo, es necesario distinguir qué tipo de tareas hay en los STR.

4.2.1. Tipos de tareas de tiempo real

Antes de ver los distintos tipos de tareas de tiempo real que pueden aparecer en un sistema, es necesario introducir ciertos conceptos básicos sobre la ejecución de tareas dentro de un STR:

- La **activación** de una tarea es un instante de tiempo desde el cual será necesario que se complete (comience y termine de ejecutar) dicha tarea antes de un determinado tiempo máximo.
- El **tiempo de respuesta máximo** (o *deadline*) es el tiempo máximo de respuesta que tiene la tarea para ejecutarse desde su última activación.

De forma general, supondremos que los tiempos de respuesta máximos son menores que el tiempo que transcurre entre dos activaciones distintas de la tarea.

¹Suponiendo que la función `sleep_for` acepta las medidas temporales en milisegundos, como es habitual.

En relación a los tipos de plazos durante los que esperamos la respuesta de una determinada tarea en un sistema de tiempo real, podemos distinguir varios tipos de tareas en los STR:

Tareas periódicas.

Son tareas cuyos instantes de activación son periódicos, con un determinado periodo T .

Tareas esporádicas.

Son tareas periódicas con instantes de activación no estrictos. Es decir, entre una activación de la tarea y la siguiente pasará al menos un periodo T , pero no necesariamente cada T instantes de tiempo tendremos una activación de la tarea.

Suelen requerir una gran urgencia cuando se activan (es decir, pequeños tiempos de respuesta máximos).

Tareas aperiódicas.

Son tareas que no tienen asociado un periodo. Por tanto, pueden activarse en cualquier momento durante la ejecución del STR.

4.2.2. Características del modelo simple de tareas

Las restricciones a considerar serán las siguientes:

- Consideramos un programa de tiempo real como un conjunto fijo de tareas (el número de tareas es constante durante toda la ejecución del programa) que se ejecutan compartiendo el tiempo de un solo procesador de forma concurrente.
- Consideraremos que todas las tareas del sistema son periódicas con periodos conocidos e independientes entre sí. No permitiremos el uso de semáforos u objetos compartidos de forma que una tarea pueda bloquear a otra.
- Todas las tareas poseen un tiempo de respuesta máximo que se considera igual a su periodo. De esta forma, una tarea está obligada a terminar completamente su ejecución antes de la siguiente activación.
- Las sobrecargas o retrasos que pueda experimentar el sistema (por ejemplo, debido a cambios de contexto) son ignorados. De esta forma, suponemos que nada impide a una tarea ejecutable obtener el procesador si en algún momento es la más prioritaria.
- Los eventos no son almacenados: si no se atienden dentro de su *deadline* se pierden.
- El tiempo máximo de cómputo de cada tarea es conocido a priori.

4.3. Atributos temporales de una tarea

Una tarea de tiempo real τ_i puede ser caracterizada mediante un conjunto específico de atributos temporales, los cuales pasamos a describir a continuación:

Prioridad (P). Es la prioridad asignada a la tarea, en el caso de considerar un algoritmo basado en pasar a ejecución las tareas más prioritarias.

Tarea (τ). El nombre de la tarea.

Instante o tiempo de activación de la tarea (t_a). Instante en el que la tarea está lista para su ejecución.

Instante o tiempo de comienzo (t_s). Instante en el que la tarea comienza a ejecutarse tras su instante de activación.

Instante o tiempo de finalización (t_f). Instante en el que la tarea finaliza su ejecución.

Instante o tiempo límite, *absolute deadline* (t_l, d). Instante de tiempo límite para la i -ésima activación de la tarea.

Periodo de ejecución (T). Su significado depende del tipo de la tarea.

Latencia (J). Intervalo de tiempo desde que se activa la tarea hasta que se ejecuta:
 $J = t_s - t_a$.

Tiempo de cómputo (c). Tiempo de cómputo de la tarea.

Tiempo de cómputo máximo (C). Tiempo de cómputo de la tarea en el peor caso posible.

Tiempo de ejecución (e). Tiempo transcurrido desde que comenzó la tarea hasta que terminó: $e = t_f - t_s$.

Tiempo de respuesta (R). Tiempo que ha necesitado la tarea para completarse desde su activación: $R = J + e$.

Plazo de respuesta máximo, *relative deadline* (D). Máximo tiempo de respuesta válido para una tarea.

Desplazamiento o fase (Φ). Primer instante de activación de la tarea.

Fluctuación relativa o *jitter* (RJ). Máxima desviación en el tiempo de comienzo entre dos activaciones sucesivas de una tarea.

Fluctuación absoluta (AJ). Máxima desviación en el tiempo de comienzo de todas las activaciones de una tarea.

Retraso o exceso, *lateness* (L). Retraso de la finalización de la tarea respecto de su tiempo límite: $L = \max\{0, t_f - t_l\}$.

Holgura (H). Tiempo máximo que una tarea puede permanecer activa sin entrar a ejecución dentro del plazo de respuesta máximo: $H = D - c$.

4.4. El problema de la planificación

El principal problema a resolver en este Capítulo será el de la planificación de actividades, en el que buscamos asignar recursos y tiempo a actividades, con el fin de maximizar una función objetivo. En nuestro caso, el principal recurso a asignar es el tiempo del procesador. Será también de nuestro interés determinar a-priori si una tarea es capaz de completar su ejecución antes de que se alcance su tiempo límite.

4.4.1. Ejecutivo cíclico

Podemos resolver el problema de la planificación de forma sencilla mediante el uso de un **ejecutivo cíclico**, un método de planificación fija de tareas que resuelve de forma sencilla el problema de la planificación de las mismas.

La planificación cíclica de las tareas se basa en una estructura fija de activación de las tareas periódicas. Hablaremos del **plan principal** de las tareas, que es un ciclo con duración un **hiperperiodo**. Un hiperperiodo de varias tareas $\tau_1, \tau_2, \dots, \tau_n$ viene dado por el mínimo común múltiplo de sus periodos (considerando que la unidad que cuenta el tiempo es entera):

$$T_M = mcm(T_1, T_2, \dots, T_n)$$

El ejecutivo cíclico consiste en definir cómo el plan principal, que indica cómo se ejecutan las tareas dentro de un hiperperiodo. Para ello, lo descompondremos en ciclos secundarios de igual duración, T_S . De esta forma:

- Lo que hacemos es establecer un plan principal explícito de las tareas sin acceso concurrente al procesador, estableciendo de una vez cómo se entrelazan las tareas en cualquier ejecución del programa, pero pudiendo variar dentro de cada ciclo secundario.
- Cada tarea se tendrá que ejecutar completamente una sola vez dentro de cada repetición de su periodo.
- Si una tarea se inicia dentro de un ciclo secundarios, esta debe terminar su ejecución dentro de dicho ciclo.

Ciclo secundario

Dadas n tareas, sabemos calcular ya su hiperperiodo, pero el valor del periodo secundario no es fijo, sino que podemos elegir el que queramos. Sin embargo, este ha de cumplir ciertas restricciones para ser un buen ciclo secundario:

- Como hemos dicho antes, el ciclo secundario divide el hiperperiodo en varias partes iguales, por lo que T_S ha de ser un divisor de T_M .
- Como cada tarea debe terminar antes de que finalice un ciclo secundario si comienza en el mismo y todas las tareas han de ejecutarse, el tiempo de cómputo de la tarea de mayor duración debe poder contenerse dentro de un ciclo secundario:

$$\max\{C_1, C_2, \dots, C_n\} \leq T_S$$

- Como cada tarea ha de poder ejecutarse completamente una sola vez dentro de cada repetición de su periodo, la duración del ciclo secundario no podrá superar el menor periodo de las tareas:

$$T_S \leq \min\{T_1, T_2, \dots, T_n\}$$

De esta forma, cualquier tiempo que cumpla que sea divisor de T_M y que:

$$\max\{C_1, C_2, \dots, C_n\} \leq T_S \leq \min\{T_1, T_2, \dots, T_n\}$$

Es un candidato a ser considerado como duración del periodo secundario.

Ejemplo. Si disponemos de las siguientes 5 tareas a planificar:

τ_i	T_i	C_i
A	25	10
B	25	8
C	50	5
D	50	4
E	100	2

Podemos tratar de crear un ejecutivo cíclico que resuelva el problema de la planificación de dichas tareas. Para ello, calculamos el hiperperiodo de todas ellas:

$$T_M = mcm(25, 50, 100) = 100$$

Posteriormente, elegimos la duración del ciclo secundario, que ha de ser divisor de 100 y ha de cumplir que:

$$\max\{2, 4, 5, 8, 10\} = 10 \leq T_S \leq 25 = \min\{25, 50, 100\}$$

Por tanto, nuestro conjunto de candidatos a ser ciclo secundario es $\{10, 15, 20, 25\}$. Elegimos $T_S = 25$, por lo que tendremos que crear un plan principal dividido en 4 ciclos secundarios de duración 25.

Para repartir las tareas entre estos ciclos secundarios, como las tareas A y B tienen periodos de 25, estas deberán ejecutarse en todos los ciclos secundarios. Una vez resueltas las tareas A y B , como C y D tienen periodo 50, han de ejecutarse cada 2 ciclos secundarios, por lo que (por ejemplo, y porque los tiempos de cómputo nos lo permiten) podemos ejecutar la tarea C en el primer y tercer ciclo y la tarea D en el segundo y cuarto ciclo. Finalmente, introducimos la tarea E una vez en algún lugar del plan principal. Por tener $C = 2$, nos da una gran flexibilidad para ejecutarla, pudiendo ponerla por ejemplo en el segundo ciclo secundario. El ejecutivo cíclico podemos observarlo en la Figura 4.1.

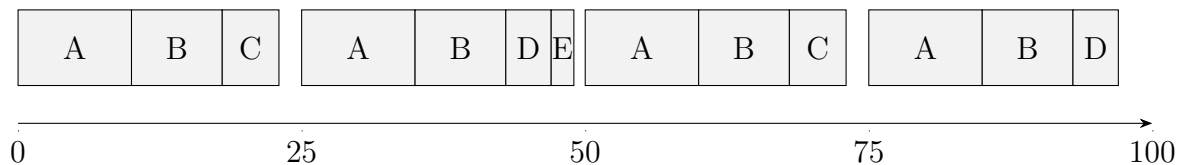


Figura 4.1: Plan principal de planificación de las tareas.

Propiedades

Algunas propiedades del ejecutivo cíclico son:

- No hay concurrencia en la ejecución: cada ciclo secundario es una secuencia de llamadas a procedimientos, por lo que no se necesita un núcleo de ejecución multitarea o un sistema operativo de tiempo real.
- Los procedimientos acceden de forma secuencial a los datos que pudieran compartirse entre las tareas, por lo que no se necesitan mecanismos de exclusión mutua.
- No hace falta analizar el comportamiento temporal del programa, ya que los plazos se cumplirán siempre por la forma en la que hemos construido el plan principal.

Implementación

Una ventaja de los ejecutivos cíclicos es que una vez diseñados son fáciles de implementar en cualquier lenguaje de programación, tal y como vemos en la Figura 4.2, donde hemos implementado el ejecutivo del ejemplo superior en C++.

```
1 void Ejecutivo(){
    const milliseconds tmp_secundario(25);
    const int nciclos = 4, // nº de ciclos secundarios
        siguiente_instante = clock::now();
5    int frame = 0; // nº del siguiente ciclo secundario

    while(true){
        for(frame = 1; frame <= nciclos; frame++){
            switch(frame){
10             case 0: A(); B(); C(); break;
                case 1: A(); B(); D(); E(); break;
                case 2: A(); B(); C(); break;
                case 3: A(); B(); D(); break;
            }
15
            siguiente_instante += tmp_secundario;
            sleep_until(siguiente_instante);
        }
20 }
}
```

Figura 4.2: Implementación del ejecutivo cíclico.

Problemas

Pese a la sencillez del ejecutivo cíclico, este presenta numerosos problemas:

- Presenta dificultades para incorporar tareas con periodos largos.
- Las tareas esporádicas son difíciles de tratar.

- El plan cíclico del proyecto es difícil de construir si los periodos son de diferentes órdenes de magnitud y el número de ciclos secundarios se hace grande.
- Es poco flexible y difícil de mantener, ya que cada vez que una tarea cambia hay que rehacer la planificación entera.

Por tanto, la solución de la planificación de tareas mediante ejecutivo cíclico es insatisfactoria, por no permitirnos una flexibilidad a la hora de considerar tareas o realizar modificaciones sobre el mismo.

De esta forma, buscamos sistemas que resuelvan el problema de la planificación de forma distinta, sistemas que funcionen como los que antes comentamos: que sean capaces de asignar prioridades a tareas y que la planificación en CPU dependa exclusivamente de dichas prioridades.

4.4.2. Algoritmos de planificación de tareas

Si consideramos otro enfoque de vista en la planificación de tareas de forma que no busquemos determinar cuándo han de ejecutarse las tareas, sino establecer criterios a las tareas para determinar cuándo pueden ejecutarse (como la asignación de una prioridad a cada tarea), nos encontramos ante la *planificación con prioridades*, que es la que triunfa dentro de los sistemas de tiempo real, ya que consigue resolver los problemas de un ejecutivo cíclico.

Para determinar la planificabilidad de un conjunto de tareas (si es posible una organización de todas ellas de forma que ninguna incumpla su *deadline*), es necesario utilizar un *esquema de planificación de tareas*, el cual ha de contener:

- Un algoritmo para ordenar el acceso de las tareas a los recursos del sistema.
- Una forma de predecir el comportamiento del sistema en el peor de los casos, para determinar si dicho conjunto de tareas es planificable en todos los casos.

Podemos encontrarnos con distintos esquemas de planificación de tareas de STR:

- Hablaremos de sistemas de planificación **estáticos** cuando el orden de planificación de las tareas sea fijo y puede ser determinado a-priori.
- Por otra parte, hablaremos de sistemas de planificación **dinámicos** cuando la prioridad de las tareas varíe a lo largo de la ejecución del programa.

Supondremos siempre un esquema de planificación expulsivo basado en prioridades. Es decir, a cada tarea le asignaremos una prioridad, de forma que siempre que haya una tarea lista para ejecutar con prioridad mayor a la tarea que se está ejecutando actualmente en el procesador, esta será desplazada por la tarea de mayor prioridad.

De esta forma, el problema de la planificación se reduce a saber asociar a cada tarea una prioridad distinta. La prioridad es un número positivo y, por convención, se consideran mayores aquellas prioridades con menor valor.

Prioridades asignadas de forma estática

Dentro de los sistemas estáticos de prioridades podemos encontrarnos con dos algoritmos a destacar:

Cadencia monótona, Rate Monotonic Scheduling (RMS).

Asigna las prioridades de forma directamente proporcional a los periodos de las tareas, de forma que a las tareas con menor periodo les asigna un valor de prioridad más bajo (es decir, una mayor prioridad).

Plazo de respuesta monótono, Deadline Monotonic Scheduling (DMS).

Asigna las prioridades de forma directamente proporcional a los *deadlines* de las tareas.

De esta forma, si consideramos el modelo simple de tareas, el comportamiento de RMS es idéntico al de DMS, por ser $D = T$.

Prioridades asignadas de forma dinámica

Podemos considerar también algoritmos dinámicos de asignación de prioridades, de forma que las prioridades de las tareas cambien durante la ejecución del sistema, algoritmos como:

Earliest Deadline First (EDF).

Asigna la mayor prioridad a la tarea con el *deadline* más próximo.

Least Laxity First (LLF).

Asigna la mayor prioridad a la tarea con menor holgura.

4.4.3. Criterios de planificabilidad de tareas

A continuación, nos centraremos en cómo determinar antes de la ejecución del algoritmo si un conjunto de n tareas es planificable o no. Para ello, supondremos una ejecución WCET (*worst case execution time*) y dispondremos de criterios que nos permitan predeterminar el comportamiento del algoritmos frente a dicho conjunto de tareas sin tener que determinar las prioridades para cada tarea.

Definición 4.1 (Factor de utilización). Dadas n tareas $\tau_1, \tau_2, \dots, \tau_n$, de forma que de cada una conocemos su tiempo máximo de cómputo (C_i) y su periodo (T_i), se define el tiempo máximo de utilización de la CPU por parte de dichas n tareas como:

$$U = \sum_{i=1}^n \frac{C_i}{T_i}$$

Una vez definido el factor de utilización de la CPU por parte de un conjunto de tareas a planificar, mostramos un primer test ideado por Liu y Layland, que nos marca una condición suficiente para que un conjunto de n tareas sea planificable por el algoritmo RMS:

Teorema 4.1 (Test I de Liu y Layland). *Dadas n tareas de las que conocemos sus atributos temporales y que suponemos que se planifican por el algoritmo RMS, este conjunto de tareas será planificable para cualesquiera desfases iniciales si el factor de utilización de la CPU por parte de dichas tareas no supera la constante de utilización máxima prefijada:*

$$U \leq U_0(n) = n \cdot \left(2^{\frac{1}{n}} - 1\right)$$

Notemos que se trata de una condición suficiente, por lo que si un conjunto de n tareas cumple que $U \leq U_0(n)$ entonces podemos garantizar que será planificable por RMS, pero si $U > U_0(n)$ no podemos asegurar nada. En esta situación, será necesario realizar un diagrama de Gantt donde simulemos una planificación de las tareas por RMS. Dicho digrama solo será necesario realizarlo con longitud un hiperperiodo, de forma que todos los desfases iniciales sean 0. Además, en la Tabla 4.1 mostramos los diferentes valores que puede tomar la constante $U_0(n)$ en función de n .

n	% utilización
1	100 %
2	82.85 %
3	78 %
4	75.7 %
\vdots	\vdots
∞	69.3 %

Tabla 4.1: Valores de $U_0(n)$.

Como podemos ver, siempre que un conjunto de tareas no supere la constante de $0,693 = \ln(2)$, el conjunto de tareas será planificable por RMS. Notemos que esta condición significa que el conjunto de tareas desperdicia más del 30 % del uso de la CPU, por lo que no se trata de un buen criterio para comprobar si un conjunto de tareas no es planificable por RMS, ya que puede haber conjuntos de tareas con factores de utilización mayores a $U_0(n)$ que sí sean planificables por RMS pero que el test de Liu y Layland falle, como mostramos en los siguientes ejemplos.

Ejemplo. Dadas las tareas τ_1 , τ_2 y τ_3 cuyos atributos temporales podemos observar en la siguiente tabla (considerando que $D_i = T_i$ en cada caso):

	C	T
τ_1	4	16
τ_2	5	40
τ_3	32	80

Si calculamos el factor de utilización de la CPU para dichas tareas, tenemos que:

$$U = \sum_{i=1}^3 \frac{C_i}{T_i} = \frac{4}{16} + \frac{5}{40} + \frac{32}{80} = 0,775 \leq 0,779 = 3 \cdot \left(2^{\frac{1}{3}} - 1\right) = U_0(3)$$

Por tanto, si aplicamos el primer test de Liu y Layland, tenemos que las tres tareas son planificables por RMS.

Ejemplo. Si ahora consideramos las tareas τ_1 , τ_2 y τ_3 con los siguientes atributos temporales:

	C	T
τ_1	10	40
τ_2	20	60
τ_3	20	80

Si calculamos el factor de utilización de la CPU y comprobamos el test de Liu y Layland:

$$U = \sum_{i=1}^3 \frac{C_i}{T_i} = \frac{1}{4} + \frac{1}{3} + \frac{1}{4} = 0,83 > 0,779 = U_0(3)$$

El test no nos dice nada, por lo que tenemos que crear el cronograma para una duración de:

$$\text{mcm}(40, 60, 80) = 240$$

Y observar el comportamiento de las tareas con un desfase inicial de 0:

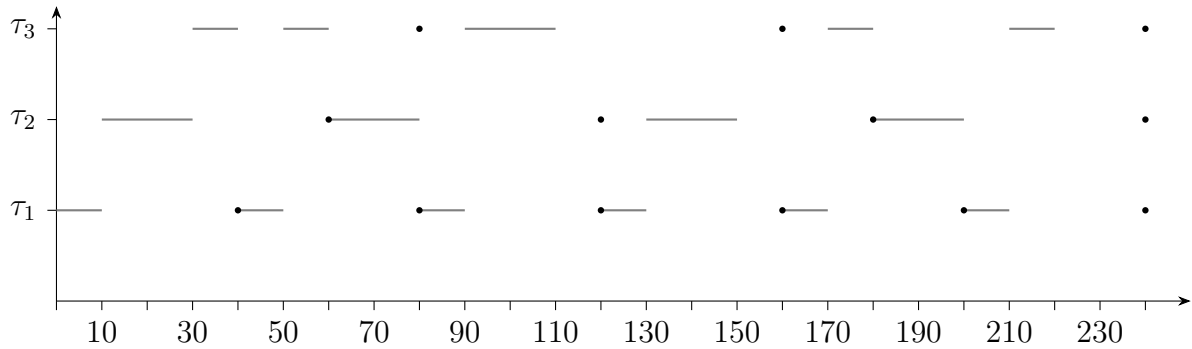


Figura 4.3: Diagrama de Gantt para las tareas por RMS.

En la Figura 4.3 podemos ver la ejecución de las tareas por RMS, donde hemos hecho visibles los *deadlines* de cada tarea, mediante puntos. Como todos los *deadline* se cumplen, el conjunto de tareas es planificable por RMS, a pesar de que el test de Liu y Layland no nos aportara información sobre este caso.

Una vez vista una condición suficiente para la planificabilidad en RMS, mostramos otro resultado, también de Liu y Layland, que nos da una condición necesaria y suficiente para que un conjunto de tareas sea planificable por EDF:

Teorema 4.2 (Test II de Liu y Layland). *Dadas n tareas de las que conocemos sus atributos temporales y que suponemos que se planifican por el algoritmo EDF, este conjunto de tareas será planificable si y solo si el factor de utilización de la CPU por parte de dicho conjunto no supera el 100 % de la CPU. Es decir:*

$$U \leq 1$$

Recordemos que el algoritmo de planificación EDF era un algoritmo dinámico, por lo que la prioridad de las tareas varía a lo largo de la ejecución del mismo. Es por esto, que siempre que un algoritmo sea planificable por un algoritmo de planificación estática (como lo es RMS), entonces lo será por un algoritmo de planificación dinámica (como por ejemplo EDF). De esta forma, podemos utilizar el Test II de Liu y Layland para afirmar que si el factor de utilización por parte de un conjunto de tareas es superior a 1, entonces dicho conjunto de tareas no podrá ser planificable por EDF ni por RMS.

Ejemplo. Consideramos las tareas τ_1 y τ_2 , cuyos atributos temporales podemos observar en la siguiente tabla:

	C	T
τ_1	2	5
τ_2	4	7

Si calculamos el factor de utilización de la CPU:

$$U = \sum_{i=1}^2 \frac{C_i}{T_i} = \frac{2}{5} + \frac{4}{7} = 0,971 > 0,82 = U_0(2)$$

Vemos que no pasa el primer test de Liu y Layland, por lo que no sabemos en un principio si el conjunto de tareas es o no planificable por RMS. Sin embargo, como $U \leq 1$, tenemos garantizada la planificabilidad por EDF. En la Figura 4.4 mostramos los casos de ejecución del conjunto de tareas por EDF y RMS, para observar las diferencias de los algoritmos.

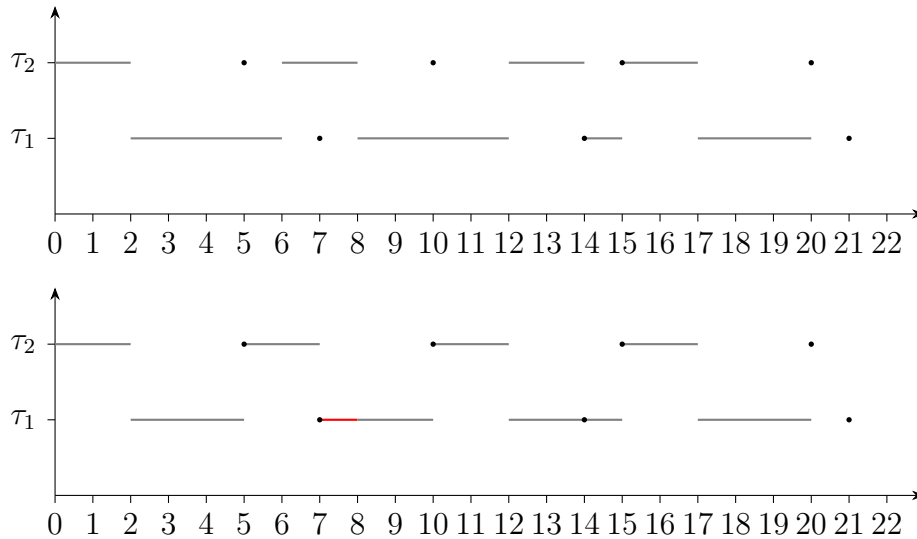


Figura 4.4: Ejecución del conjunto de tareas por EDF y RMS.

Aunque para comprobar la planificabilidad de τ_1 y τ_2 tendríamos que haber hecho el diagrama hasta $mcm(5, 7) = 35$, con hacer el diagrama hasta 22 podemos observar ya ciertas diferencias en la ejecución de los algoritmos. En primer lugar, observamos que con EDF (primera figura) no incumplimos ningún *deadline*, mientras que en la

segunda figura (RMS) sí que incumplimos el primer *deadline*, al tener que terminar de ejecutarse la primera activación de τ_1 tras $t = 7$.

Hemos visto que la planificabilidad con algoritmos dinámicos parece ser más predecible que con los algoritmos estáticos, así como que los algoritmos dinámicos nos permiten una mayor flexibilidad, ya que siempre que un conjunto de tareas sea planificable por algoritmos estáticos, lo será por algoritmos dinámicos. Sin embargo, los algoritmos estáticos tienen ventajas frente a los dinámicos, tal y como mostramos a continuación:

- Los esquemas estáticos son más sencillos y eficientes de implementar.
- Los esquemas dinámicos introducen una mayor sobrecarga al sistema.
- Los esquemas estáticos permiten incorporar otros factores que influyen en la planificación de un conjunto de tareas cuando sus prioridades no están asociadas a un tiempo límite.
- Durante sobrecarga transitoria, un esquema de planificación estático normalmente resulta ser predecible. Lo cual no ocurre necesariamente si se utiliza un esquema dinámico.

4.5. Modelo general de tareas

El modelo simple ha de ser extendido para poder incluir los requisitos de planificación de las tareas esporádicas y aperiódicas, así como los bloqueos que sufren las tareas cuando intentan obtener recursos compartidos a los que acceden en exclusión mutua. Ya que en el modelo simple de tareas se considera a éstas totalmente independientes durante toda su ejecución.

Ahora, consideraremos las siguientes restricciones sobre las tareas:

- Veremos a un programa como un conjunto fijo de tareas que comparten el tiempo de un procesador, pero pueden aparecer tareas aperiódicas o esporádicas.
- Las tareas pueden bloquearse por acceder a recursos compartidos.
- Las tareas tienen un plazo de respuesta máximo que, en general, no coincide con su periodo.
- Los retrasos debidos al sistema son ignorados.

4.6. Inversión de prioridad

Al considerar el modelo general de tareas nos encontramos con un primer problema, resultado de permitir que las tareas puedan interaccionar entre sí al acceder a recursos compartidos.

El problema que surge si una tarea permanece bloqueada esperando a que otra menos prioritaria termine de ejecutar una sección de su código, es que deja de cumplirse una de las condiciones más importantes del modelo simple de tareas: cuando una tarea tiene prioridad suficiente para ejecutarse, ha de hacerlo. Según el modelo simple, en ningún caso podría verse bloqueada una tarea, o suspendida su ejecución durante algún tiempo, ya que esto influiría en la planificación del resto de las tareas y podría llegar a ocasionar la pérdida de sus tiempos límite. Por tanto, no sería de aplicación el análisis de planificabilidad basado en el Test de Liu y Layland, para el modelo simple de tareas, sin cuantificar cómo se vería afectada la desigualdad del test con el eventual bloqueo de las tareas al acceder a recursos comunes.

Podría suceder, incluso, que la espera de la tarea más prioritaria llegase a ser arbitrariamente larga si se ejecutan continuamente tareas menos prioritarias mientras que ésta se mantiene bloqueada. En ese caso, se produciría lo que se denomina una *inversión de prioridad* que invalidaría cualquier previsión acerca de la planificabilidad de un conjunto de tareas. La inversión de prioridad no puede ser eliminada completamente si se utilizan objetos protegidos en los programas de tiempo real, pero los efectos adversos sobre la planificación de las tareas más prioritarias pueden ser minimizados, haciendo que el bloqueo derivado del acceso a dichos objetos sea siempre un tiempo acotado y medible en cualquier ejecución de la aplicación.

Ejemplo. A continuación podemos observar un claro ejemplo de inversión de prioridad (el cual usaremos a lo largo de los métodos para remediarla), que sucede sobre el conjunto de tareas de la Tabla 4.2, donde la tarea τ_1 comparte una sección crítica con las tareas τ_4 (comparte **s1**) y τ_2 (comparte **s2**).

	C_i	P_i	Φ_i
τ_1	5	1	4
τ_2	4	2	2
τ_3	2	3	2
τ_4	6	4	0

Tabla 4.2: Conjunto de tareas para el ejemplo.

Entre $t = 0$ y $t = 2$ la única tarea activa es τ_4 , que adquiere acceso a la sección crítica **s1**. En $t = 2$ se activan tanto τ_3 como τ_2 , pasando a ejecución τ_2 por ser la más prioritaria (y quitándole la CPU a τ_4), donde adquiere su sección crítica, **s2**. En $t = 4$ se activa τ_1 , quien adquiere la CPU por ser la tarea más prioritaria.

En el instante en el que τ_1 se disponga a adquirir acceso a la sección crítica **s1**, se verá bloqueada, ya que estaba siendo usada por τ_4 . Entrará ahora en la CPU τ_2 , quien terminará de usar su sección crítica. Como τ_1 está bloqueada hasta que τ_4 libere la sección crítica **s1**, entrará en CPU τ_3 , que ejecutará su código. Finalmente, entrará en sección crítica τ_4 , que ejecutará la sección crítica hasta liberarla, momento en el cual τ_1 se quedará con la CPU, hasta terminar su ejecución y devolverla a τ_4 , quien también terminará su ejecución.

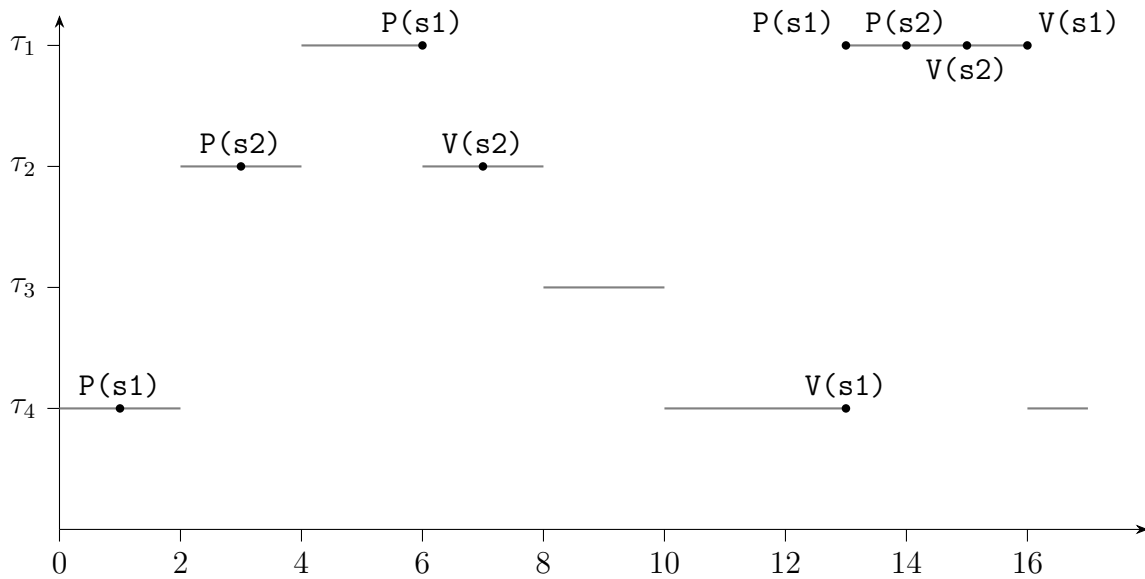


Figura 4.5: Ejecución de las tareas.

En la Figura 4.5 podemos observar el comportamiento de las tareas, donde observamos una inversión de prioridad, al ser las tareas τ_2 , τ_3 y τ_4 menos prioritarias que τ_1 y estar ejecutándose entre $t = 6$ y $t = 13$ en la CPU, cuando la tarea que se debería estar ejecutando es τ_1 .

A continuación, veremos tres técnicas que permiten reducir el efecto de la inversión de prioridad (cada una mejora la anterior). Si bien estas técnicas son menos perjudiciales que no hacer nada al respecto, no consiguen solventar del todo el problema de la inversión de prioridad, pero son capaces de disminuir los retrasos introducidos por el problema de la inversión.

4.6.1. Sección crítica no expulsable

Es el protocolo más simple que puede imaginarse para intentar solventar el problema de la inversión de prioridad. Consiste en que durante el tiempo en el que una tarea hace uso de un recurso compartido, esta no puede ser expulsada de la CPU por una tarea de mayor prioridad. En este caso, la tarea de mayor prioridad ha de esperar a que la tarea de menor prioridad finalice la sección crítica, momento en el que podrá expulsarla de la CPU y comenzar a ejecutar su código.

Ejemplo. Sobre el ejemplo anteriormente visto en el que presentábamos el problema de la inversión de prioridad podemos ahora establecer el protocolo de “sección crítica no expulsable”, para ver los efectos sobre el comportamiento de ejecución de las tareas que considerábamos anteriormente. Estos cambios pueden verse en la Figura 4.6.

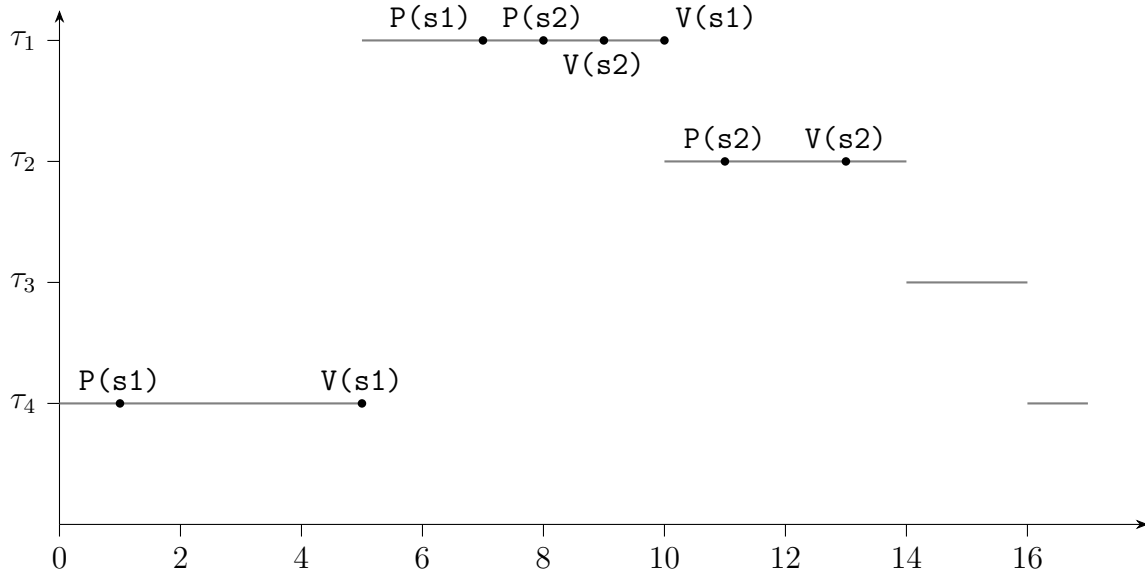


Figura 4.6: Ejecución de las tareas con sección crítica no expulsable.

Ahora, en $t = 2$ y $t = 4$ la tarea τ_4 no se ve desplazada de la CPU porque haya tareas más prioritarias, ya que se encuentra dentro de su sección crítica. En el momento en el que τ_4 libere su sección crítica (sucede en $t = 7$), entonces puede ser desplazada de la CPU por la tarea más prioritaria del momento, como lo es τ_1 . Tras esto, la ejecución de las tareas se realiza de forma normal gracias a las prioridades asignadas previamente.

Tiempo de bloqueo

Resulta que con este protocolo podemos seguir aplicando el Test I de Liu y Layland para ver si un conjunto de tareas es o no planificable. Para ello, deberemos ver para cada una de las tareas que estemos considerando cual es el tiempo máximo de bloqueo debido a tareas menos prioritarias². Este tiempo extra recibe el nombre de *factor de bloqueo*, al que notaremos por B_i para la tarea i -ésima. De esta forma, el nuevo tiempo máximo de cómputo que consideraremos para cada tarea será:

$$C'_i = C_i + B_i \quad \forall i \in \{1, \dots, n\}$$

El factor de bloqueo se va a calcular considerando el peor caso posible de planificación, es decir, el mayor bloqueo que una tarea prioritaria podría experimentar debido a que comparta alguna sección crítica con tareas menos prioritarias que ella.

En el caso de la sección crítica no expulsable, como cada tarea solo puede ser bloqueada como máximo una vez por una tarea menos prioritaria que ella (este suceso solo puede darse si la tarea menos prioritaria se activa y adquiere la sección crítica antes de que se active la tarea más prioritaria, de forma que esta se active mientras la otra esté ejecutando la sección crítica), entonces el factor de bloqueo para la i -ésima tarea será la máxima duración de ejecución de una sección crítica que

²Es decir, el tiempo extra de cómputo que se ha introducido a cada tarea por estar usando este protocolo.

comparten la tarea i y la tarea j , siendo la tarea j de menor prioridad³ que la tarea i .

A continuación, para cada protocolo para evitar la aparición de la inversión de prioridad, calcularemos el factor de bloqueo en cada caso, con el fin de poder aplicar el test de Liu y Layland en cada caso, para comprobar la planificabilidad por RMS.

4.6.2. Herencia de prioridad

Este protocolo cambia de forma dinámica las prioridades de las tareas que consideramos, aunque estemos usando un protocolo de asignación estática de prioridades como lo es RMS. En este caso, si una tarea se dispone a ejecutar una sección crítica que está siendo ejecutada (y por tanto la tarea se tendría que bloquear) por otra tarea de menor prioridad, entonces la tarea de menor prioridad “hereda” la prioridad de la tarea que será bloqueada durante la ejecución de dicha sección crítica. Tras la ejecución de la sección crítica, la tarea de mayor prioridad que quedó bloqueada esperando la finalización de la sección crítica podrá adquirir el recurso, y la tarea que estaba ejecutando la sección crítica recuperará su prioridad original.

Ejemplo. Recuperando el ejemplo anterior, si consideramos ahora el protocolo de herencia de prioridad sobre la ejecución de las tareas anteriores, obtenemos el comportamiento de la Figura 4.7.

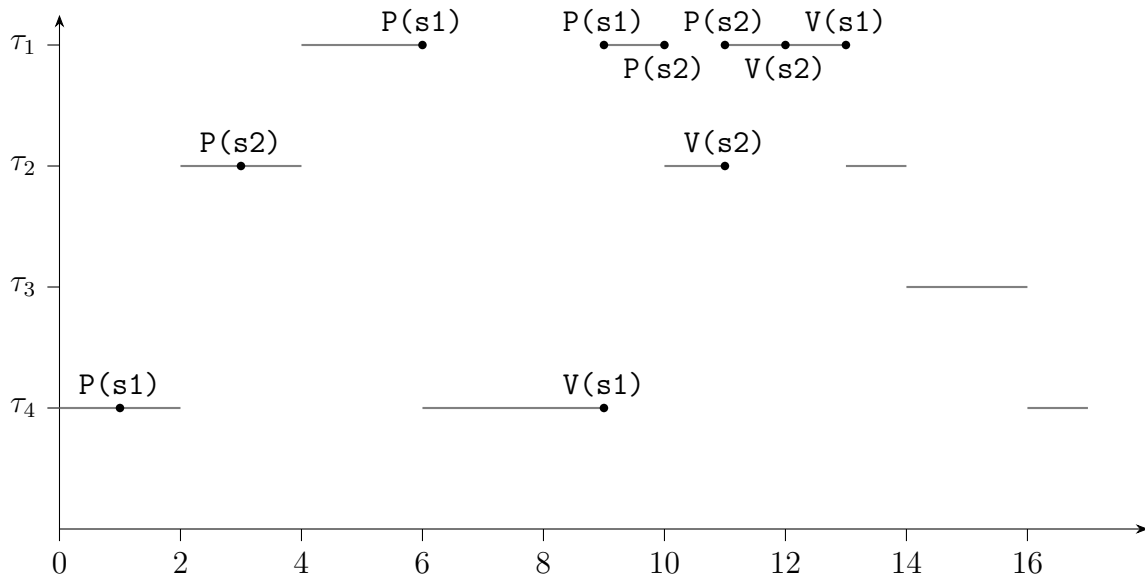


Figura 4.7: Ejecución de las tareas con herencia de prioridad.

El comportamiento hasta $t = 6$ es el mismo que teníamos sin ningún protocolo para evitar la inversión de prioridad. Sin embargo, cuando τ_1 intente ejecutar $P(s1)$, la tarea τ_4 heredaré la prioridad de τ_1 y se ejecutará con prioridad 1 mientras que tenga la sección crítica $s1$ en uso (tras su liberación, volverá a prioridad 4). Un comportamiento similar sucede en $t = 10$ cuando τ_1 intenta ejecutar $P(s2)$: τ_2 hereda la prioridad de τ_1 mientras que ejecute la sección crítica $s2$. Una vez que τ_1

³Es decir, mayor número de prioridad.

disponga de todas las secciones críticas a su disposición, será capaz de completar su ejecución y el resto de las tareas entrarán en la CPU en el orden que indica la prioridad de cada una.

El ejemplo superior además de darnos un caso particular de ejecución de un conjunto de tareas usando el algoritmo de herencia de prioridad nos muestra algo más, y es que con este protocolo las tareas pueden sufrir ahora dos tipos de bloqueos:

- Bloqueos directos, como el que sufre τ_1 por tener que esperar a las secciones críticas s_1 y s_2 por parte de τ_4 y τ_2 .
- Bloques indirectos, como los que sufren τ_2 y τ_3 , al quedar bloqueadas y ejecutarse τ_4 , ya que heredó la prioridad de τ_1 .

Estos tipos de bloqueos serán necesarios de entender a la hora de calcular el factor de bloqueo para este protocolo.

Además, el protocolo de herencia de prioridad no evita ni el interbloqueo de las tareas en el acceso a recursos ni los bloqueos encadenados o transitivos.

Tiempo de bloqueo

Con este protocolo, una tarea solo puede verse bloqueada un número limitado de veces por otras menos prioritarias, de forma que:

- Si una tarea tiene definidas en su código M secciones críticas, entonces el número máximo de veces que puede verse bloqueada durante su ejecución es M .
- Si hay solo $N < M$ tareas menos prioritarias que la que estamos considerando, entonces el número máximo de bloqueos que puede experimentar la tarea más prioritaria se reduce a N .

Para calcular el factor de bloqueo B_i de una tarea τ_i hay que establecer qué secciones críticas pueden estar en ejecución por otras tareas menos prioritarias cuando se active la primera y cuáles de estas pueden bloquearla. El factor de bloqueo para este protocolo suele ser difícil de calcular, por lo que aquí plantearemos una cota superior del tiempo de bloqueo, que se calculará de la siguiente forma (considerando que la tarea i es más prioritaria que la j si $i < j$):

- En primer lugar, calculamos B_i^l , bloqueos debidos a tareas menos prioritarias que i .
- Posteriormente, calculamos B_i^s , bloqueos debidos a todas las secciones críticas a las que accede la tarea τ_i .

Finalmente, consideramos que el factor de bloqueo para la i -ésima tarea es:

$$B_i = \min\{B_i^l, B_i^s\}$$

4.6.3. Techo de prioridad

El protocolo de herencia de prioridad tenía algunas contrapartidas, como por ejemplo que no está libre de interbloqueos o cadenas de bloqueos; así como que la estimación del tiempo que puede quedar bloqueada una tarea es difícil de calcular y puede ser muy pesimista en algunos casos.

Es por esto que se plantean los protocolos de techo de prioridad, entre los que destacamos:

- El protocolo original de techo de prioridad (OCPD).
- El protocolo de techo de prioridad inmediato (ICPD).

Aunque el que trataremos ahora será el segundo, que es la mejor solución al problema de la inversión de prioridad, y es la que usa en Sistemas Operativos POSIX.

Ahora, a cada sección crítica le asignamos una prioridad, a la que llamaremos *techo de prioridad* de la sección crítica. Este valor será igual a la prioridad más grande de la tarea que en algún momento (durante la ejecución de su código) hace uso de esta sección crítica.

Cuando una tarea quiera acceder a una sección crítica, solo podrá acceder a ella en caso de que esté libre y de que su prioridad sea estrictamente mayor que el techo de prioridad de cualquier otra sección crítica en uso.

Ejemplo. Recuperando por última vez el ejemplo que venimos usando en todos los protocolos para evitar la inversión de prioridad, en la Figura 4.8 podemos observar ahora el comportamiento de las tareas si usamos el protocolo de techo de prioridad.

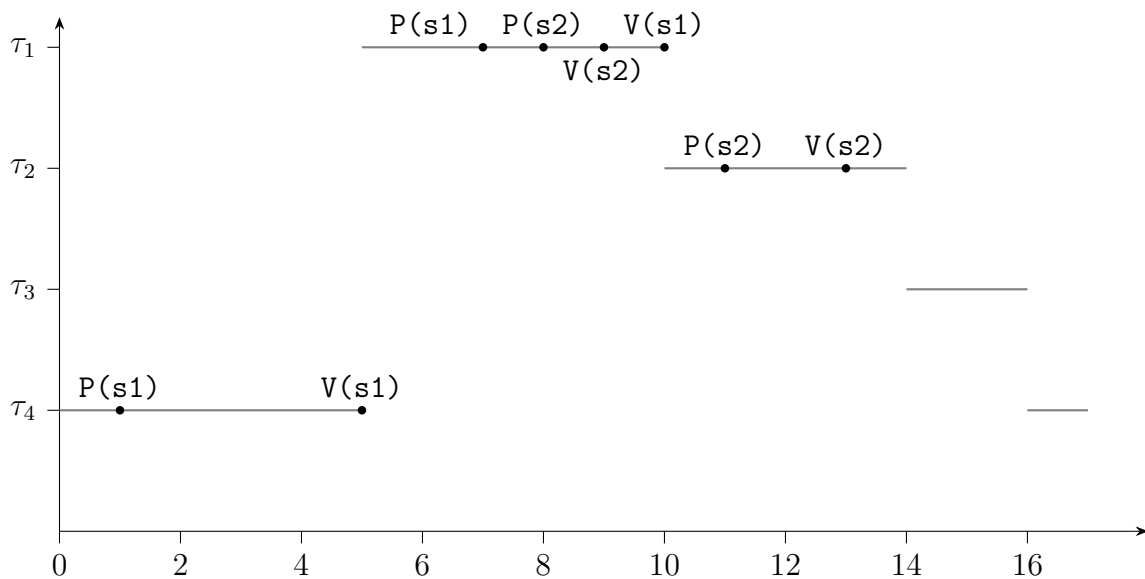


Figura 4.8: Ejecución de las tareas con techo de prioridad.

Ahora, debemos tener en cuenta que la sección crítica $s1$ tiene un techo de prioridad de 1 (ya que es usada por la tarea τ_1) y que la sección crítica $s2$ también

tiene el mismo techo de prioridad. En $t = 2$ se activan las tareas τ_2 y τ_3 , siendo τ_2 la más prioritaria, por lo que intenta acceder a la CPU. Sin embargo, como la prioridad de τ_2 (2) no es superior al techo de prioridad de $s1$ (1), τ_2 se ve bloqueada. En $t = 4$ se activa τ_1 , cuya prioridad no es mayor que el techo de prioridad de $s1$, por lo que también se ve bloqueada hasta que termina $s1$, momento en el que τ_1 entra a ejecución y ya se sigue el orden correspondiente por prioridades de las tareas en el acceso a la CPU.

Tiempo de bloqueo

Al igual que sucedía con el protocolo de sección crítica no expulsable, como cada tarea solo puede sufrir un único bloque inicial por una tarea de menor prioridad, entonces el factor de bloqueo para la tarea i -ésima será igual a la duración de la sección crítica más larga a la que accedan tareas de prioridad inferior y que posean un techo de prioridad mayor o igual que la prioridad de la tarea i .

4.7. Planificación de tareas aperiódicas

Otro problema al que nos enfrentamos con el modelo general de tareas por el que no teníamos que preocuparnos con el modelo simple de tareas es la aparición de las tareas aperiódicas, tareas que ahora debemos tener en cuenta a la hora de planificar las tareas periódicas, tratando de darles servicio.

En esta sección, veremos tres formas de introducir las tareas aperiódicas en juego para poder planificarlas junto a las tareas periódicas que ya sabemos planificar.

4.7.1. Servicio de procesamiento en segundo plano

Una primera forma simple de planificar tareas aperiódicas en un sistema de tiempo real junto con un conjunto de tareas periódicas que ya hemos planificado consiste en asignarle a dichas tareas aperiódicas las menores prioridades del sistema, de tal forma que la introducción de estas tareas no modifica la planificabilidad de las tareas periódicas que anteriormente considerábamos, ya que las tareas aperiódicas serán ejecutadas en los “huecos libres” que deja el procesador al ejecutar las tareas periódicas.

Ejemplo. Consideramos las tareas periódicas τ_1 y τ_2 , así como una tercera tarea aperiódica, J , podemos observar los atributos temporales de las tareas periódicas en la siguiente tabla:

	C	T	Φ
τ_1	1.5	3.5	2
τ_2	0.5	6.5	0

Para este ejemplo, consideramos que J solo se activa una única vez, en $t = 2,8$, y que tiene un tiempo de cómputo máximo $C_J = 1,7$.

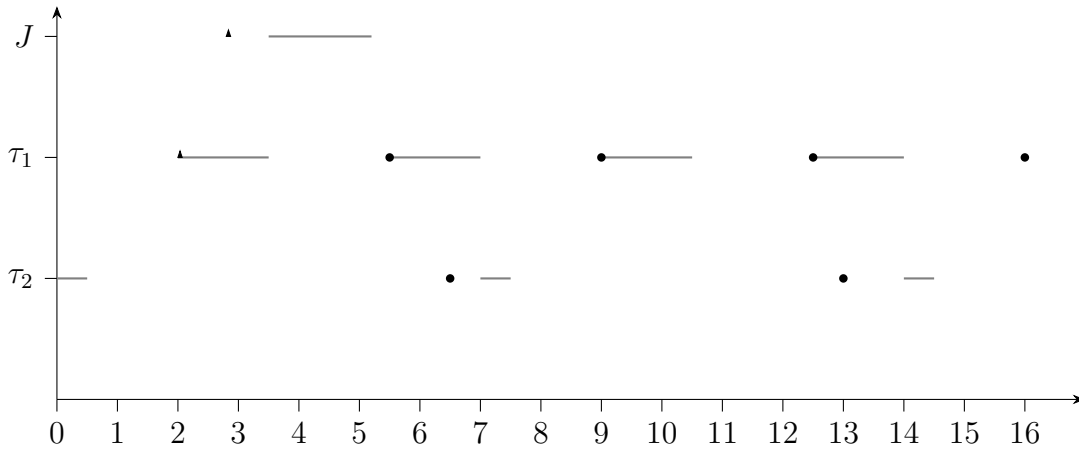


Figura 4.9: Planificación de tareas usando procesamiento en segundo plano.

Como podemos ver en la Figura 4.9, la planificabilidad de las tareas τ_1 y τ_2 no se ve afectada por la introducción de la tarea J si consideramos un procesamiento en segundo plano de tareas aperiódicas.

4.7.2. Servicio de peticiones utilizando una tarea sondeante

En este caso, se añade una *tarea sondeante* a las tareas periódicas de la aplicación (no tiene por qué ser una tarea más, sino puede ser lógica, por lo que no será necesario crear una tarea más). Se reservará un tamaño C_s equivalente al mayor tiempo de ejecución de las tareas aperiódicas a considerar y un periodo T_s para esta nueva tarea sondeante τ_s . Sin embargo, la prioridad de esta nueva tarea no se asignará en relación a T_s , sino que se puede asignar la prioridad deseada.

Con esta tarea sondeante, se puede aplicar el test de Liu y Layland para comprobar la planificabilidad del conjunto de tareas para RMS. Para ello, tendremos que considerar el factor de utilización U del procesador por parte de las n tareas periódicas, el factor de utilización por parte de la tarea sondeante y compararlo con la constante para $n + 1$ tareas, buscando que:

$$U + \frac{C_s}{T_s} \leq U_0(n + 1) = (n + 1) \cdot \left(2^{\frac{1}{n+1}} - 1\right)$$

Ejemplo. Ante el ejemplo anterior, consideramos ahora una resolución de la tarea aperiódica mediante una tarea sondeante τ_s , cuyos parámetros temporales podemos observar en la siguiente tabla:

	C	T	Φ
τ_1	1.5	3.5	2
τ_2	0.5	6.5	0
τ_s	1	3	0

A esta tarea le asignaremos la mayor prioridad del sistema. Ante esta situación, mostraremos en la Figura 4.10 el caso de ejecución considerando las mismas condiciones que en el caso anterior, con una activación que J en el instante $t = 2,8$, que requiere un tiempo de cómputo de 1,7.

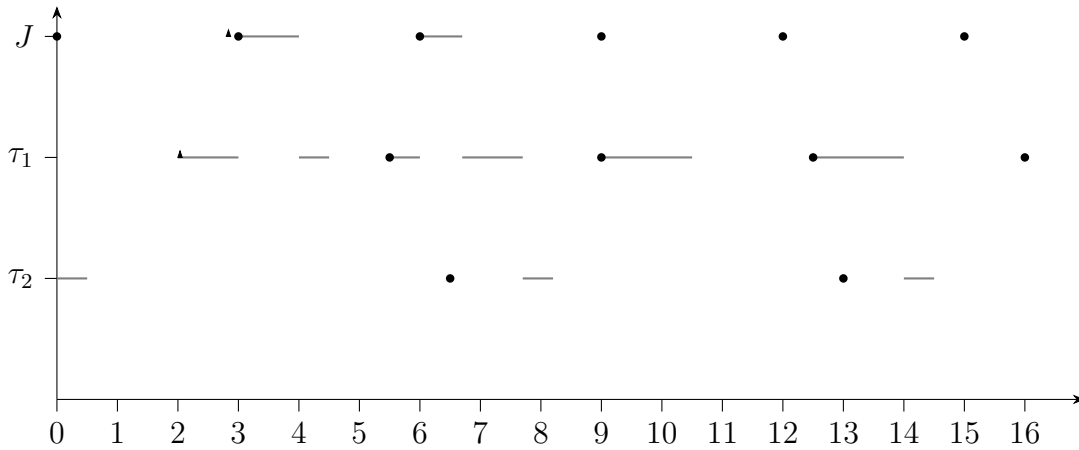


Figura 4.10: Planificación de tareas usando una tarea sondeante.

Como podemos ver, cuando no hay tareas aperiódicas que servir, la tarea sondeante no se ejecuta.

4.7.3. Servicio de peticiones utilizando un servidor diferido

En este caso, se asigna un tamaño al servidor, C_s , que se gastará en atender peticiones aperiódicas y que se rellenará a su valor máximo en cada ciclo del servidor, T_s . En este caso, se le asignará una prioridad máxima a la tarea que funcione como servidor de tareas aperiódicas, por lo que la inmediatez al atender tareas aperiódicas es característico del servidor diferido.

Los valores de C_s y de T_s se elegirán de manera que todas las tareas periódicas del sistema mantengan siempre sus tiempos límite.

Ejemplo. Si consideramos por última vez el conjunto de tareas anterior con un servidor diferido con valores de C_s y T_s iguales a los de la tarea sondeante de la sección anterior, en este caso el comportamiento observado es el de la Figura 4.11.

Como podemos ver, desde que se inicia la tarea aperiódica en $t = 2,8$ hasta el siguiente periodo del servidor diferido transcurren 0,2 unidades de tiempo, durante las cuales el tiempo del servidor disminuye, pero vuelve a aumentar en $t = 3$, por cumplirse un periodo del servidor.

Posteriormente, se gasta todo el tiempo del servidor en atender la tarea, habiendo completado un total de 1,2 unidades de tiempo de la tarea, que en total son 1,7, por lo que en el siguiente periodo del servidor diferido se deberán completar las 0,5 unidades de tiempo restantes para completar la ejecución de la tarea J .

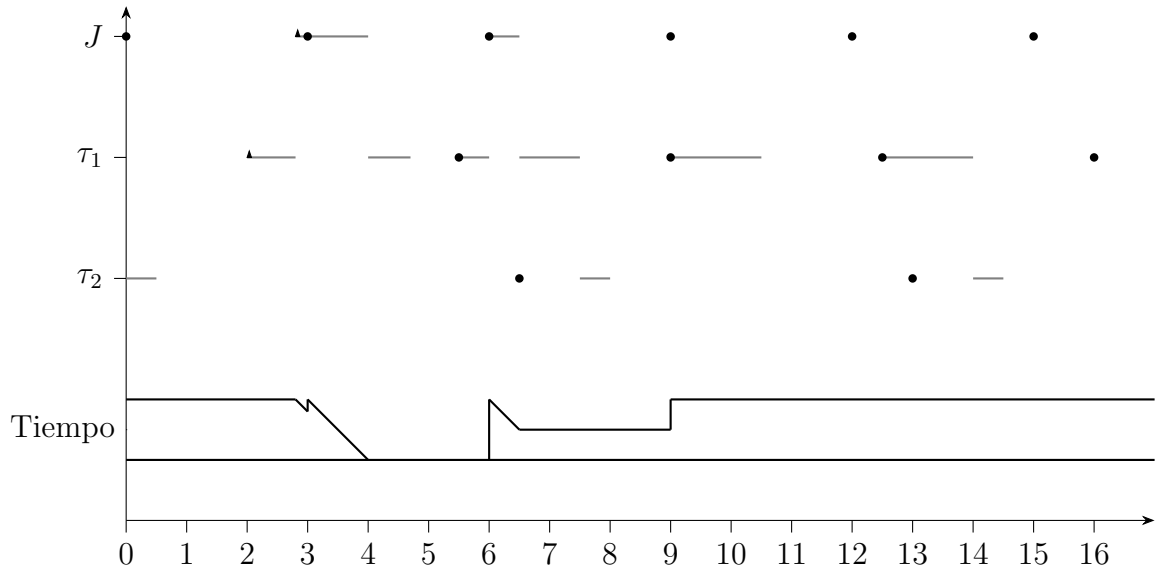


Figura 4.11: Planificación de tareas usando un servidor diferido.

Análisis de planificabilidad

Si consideramos un conjunto de n tareas periódicas $\tau_1, \tau_2, \dots, \tau_n$ y un servidor diferido de tiempos C_s y T_s , notando por U_p al factor de utilización de la CPU por parte de las n tareas periódicas y por U_s al factor de utilización de la CPU por parte del servidor diferido:

$$U_p = \sum_{i=1}^n \frac{C_i}{T_i} \quad U_s = \frac{C_s}{T_s}$$

Entonces, el conjunto de tareas periódicas junto con el servidor diferido serán planificables si se cumple la desigualdad:

$$U_p \leq n \cdot \left(\left(\frac{U_s + 2}{2U_s + 1} \right)^{\frac{1}{n}} - 1 \right)$$

Notemos que cuando $n \rightarrow \infty$ la desigualdad a comprobar resulta:

$$U_p \leq \ln \left(\frac{U_s + 2}{2U_s + 1} \right)$$

5. Relaciones de problemas

5.1. Introducción

Ejercicio 5.1.1. Considerar el siguiente fragmento de programa para 2 procesos P1 y P2: Los dos procesos pueden ejecutarse a cualquier velocidad. ¿Cuáles son los posibles valores resultantes para la variable x ? Suponer que x debe ser cargada en un registro para incrementarse y que cada proceso usa un registro diferente para realizar el incremento.

```

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

```

```

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

```

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 5.1.2. ¿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:

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

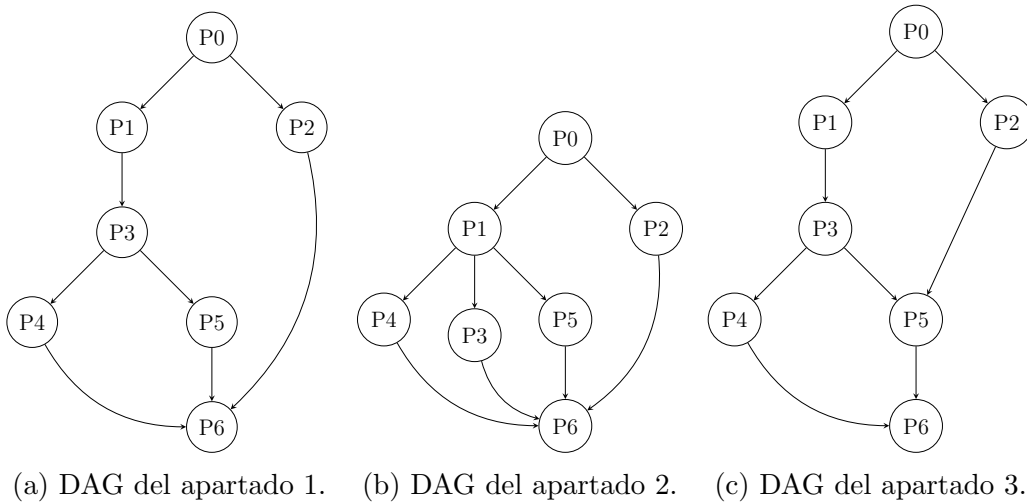


Figura 5.1: Grafos de precedencia del ejercicio 5.1.3.

Ejercicio 5.1.3. 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 5.1.

1. Grafo de precedencia de la figura 5.1a:

```

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

```

```

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

```

2. Grafo de precedencia de la figura 5.1b:

```

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

```

```

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

```

3. Grafo de precedencia de la figura 5.1c:

```

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

```

Sin embargo, no podemos hacer al 100% el DAG de la figura 5.1c, 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:

```

1  begin
    P0;
    cobegin
        P2;
5    begin
        P1; P3;
        end
    coend
    cobegin P4; P5; coend
10  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):

```

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

```

Ejercicio 5.1.4. Dados los siguientes fragmentos de programas concurrentes, obtener sus grafos de precedencia asociados:

```

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

```

```

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

```

(a) Programa 1.

(b) Programa 2.

Figura 5.4: Programas concurrentes del ejercicio 5.1.4.

1. Programa de la figura 5.4a.

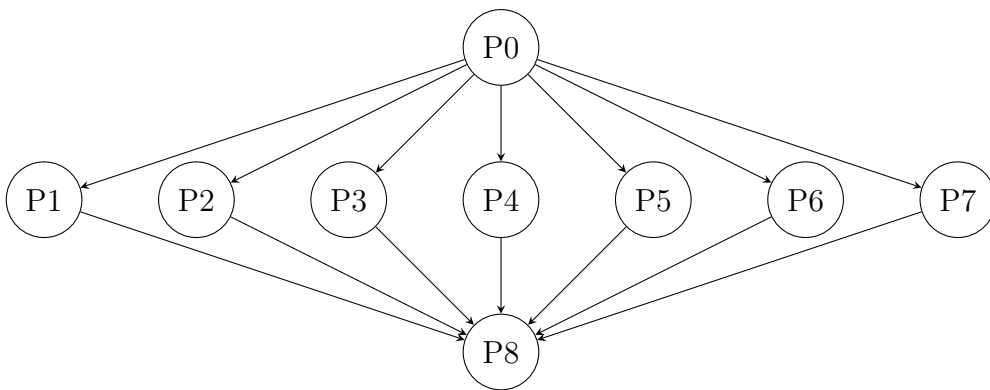


Figura 5.5: DAG para la figura 5.4a.

2. Programa de la figura 5.4b.

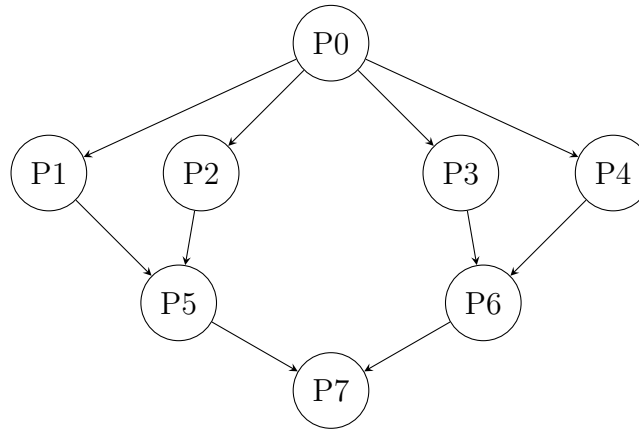


Figura 5.6: DAG para la figura 5.4b.

Ejercicio 5.1.5. 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 interfoliación 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 instrucción 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 5.1. 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.


```

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

```

Código fuente 1: Código acumulador-escritor del ejercicio 5.1.5.

Opción A			Opción B			Opción C		
Operación	n	Salida	Operación	n	Salida	Operación	n	Salida
—	k	—	—	k	—	—	k	—
$1 \rightarrow n := n+1$	$k+1$	—	$2 \rightarrow \text{write}(n)$	k	k	$2 \rightarrow \text{write}(n)$	k	k
$2 \rightarrow \text{write}(n)$	$k+1$	$k+1$	$1 \rightarrow n := n+1$	$k+1$	k	$3 \rightarrow n := 0$	0	k
$3 \rightarrow n := 0$	0	$k+1$	$3 \rightarrow n := 0$	0	k	$1 \rightarrow n := n+1$	1	k

Tabla 5.1: Tabla de opciones del ejercicio 5.1.5.

```

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

```

Código fuente 2: Procedimientos `Sort` y `Copiar` del ejercicio 5.1.6.

Ejercicio 5.1.6. Supongamos un programa concurrente en el cual hay, en memoria compartida dos vectores `a` y `b` de enteros y con tamaño par, declarados como sigue:

```

1  var a,b : array[1..2*n] of integer ; { n es una constante predefinida }

```

Queremos escribir un programa para obtener en `b` una copia ordenada del contenido de `a` (nos da igual el estado en que queda `a` después de obtener `b`). Para ello disponemos de la función `Sort` que ordena un tramo de `a` (entre las entradas `s` y `t`, ambas incluidas). También disponemos la función `Copiar`, que copia un tramo de `a` (desde `s` hasta `t`) en `b` (a partir de `o`). Estas funciones se muestran en el Código Fuente 2.

El programa para ordenar se puede implementar de dos formas:

1. Ordenar todo el vector `a`, de forma secuencial con la función `Sort`, y después copiar cada entrada de `a` en `b`, con la función `Copiar`.
2. Ordenar las dos mitades de `a` de forma concurrente, y después mezclar dichas dos mitades en un segundo vector `b` (para mezclar usamos un procedimiento `Merge`).

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

El código de la función `Merge`, disponible en el Código Fuente 4, se encarga de ir leyendo las dos mitades de `a`, en cada paso, seleccionar el menor elemento de los dos siguientes por leer (uno en cada mitad), y escribir dicho menor elemento en la siguiente mitad del vector mezclado `b`.

Llamaremos $T_s(k)$ al tiempo que tarda el procedimiento `Sort` cuando actúa sobre un segmento del vector con k entradas. Suponemos que el tiempo que (en media) tarda cada iteración del bucle interno que hay en `Sort` es la unidad (por definición).

Es evidente que ese bucle tiene $\frac{k(k-1)}{2}$ iteraciones, luego:

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

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

Código fuente 3: Procedimientos `Secuencial` y `Concurrente` del ejercicio 5.1.6.

```
1 procedure Merge( inferior, medio, superior: integer ) ;  
  { siguiente posicion a escribir en b }  
  var escribir : integer := 1 ;  
  { siguiente pos. a leer en primera mitad de a }  
5  var leer1 : integer := inferior ;  
  { siguiente pos. a leer en segunda mitad de a }  
  var leer2 : integer := medio ;  
  begin  
    { mientras no haya terminado con alguna mitad }  
10    while leer1 < medio and leer2 <= superior do begin  
      if a[leer1] < a[leer2] then begin { minimo en la primera mitad }  
        b[escribir] := a[leer1] ;  
        leer1 := leer1 + 1 ;  
      end else begin { minimo en la segunda mitad }  
15      b[escribir] := a[leer2] ;  
        leer2 := leer2 + 1 ;  
      end  
      escribir := escribir+1 ;  
    end  
20    { se ha terminado de copiar una de las mitades,  
      copiar lo que quede de la otra }  
    if leer2 > superior then  
      { copiar primera } Copiar( escribir, leer1, medio-1 );  
    else Copiar( escribir, leer2, superior ); { copiar segunda }  
25  end
```

Código fuente 4: Procedimiento `Merge` del ejercicio 5.1.6.

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:

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

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

```

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

```

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

```

1  var a, b, c : array[1..3,1..3] of
    real ;
    process CalcularFila[ i : 1..3] ;
        var j, k : integer ;
        begin
5      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
10     end
    end
end

```

(a) Procesos concurrentes para calcular por filas.

```

1  var a, b, c : array[1..3,1..3] of
    real ;
    process CalcularColumna[j : 1..3] ;
        var i, k : integer ;
        begin
5      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
10     end
    end
end

```

(b) Procesos concurrentes para calcular por columnas.

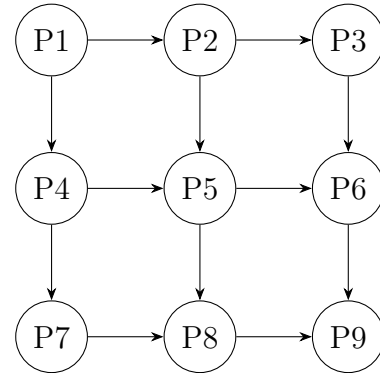
Figura 5.7: Códigos para el ejercicio 5.1.7.

```

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

```

(a) Código del ejercicio 5.1.8.



(b) DAG del ejercicio 5.1.8.

Figura 5.8: Figuras del ejercicio 5.1.8.

como que elementos distintos de c pueden escribirse simultáneamente.

Esta solución puede llevarse a cabo de dos formas (tal y como vimos en la asignatura de AC). Crearemos tres procesos, y cada uno puede calcular una fila de la matriz c (Código Fuente 5.7a) o bien cada uno puede calcular una columna de la matriz c (Código Fuente 5.7b). 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 5.1.8. Un trozo de programa ejecuta nueve rutinas o actividades (P_1, P_2, \dots, P_9), repetidas veces, de forma concurrentemente con `cobegin-coend` (ver trozo de código de la figura 5.8a), pero que requieren sincronizarse según determinado grafo (ver la figura 5.8b).

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 5.1.9. En el ejercicio 5.1.8 los procesos **P1**, **P2**, ..., **P9** se ponen en marcha usando **cobegin-coend**. Escribe un programa equivalente, que ponga en marcha

todos los procesos, pero que use declaración estática de procesos, usando un vector de procesos P , con índices desde 1 hasta 9, ambos incluidos. El proceso $P[n]$ contiene una secuencia de instrucciones desconocida, que llamamos S_n , y además debe incluir las llamadas necesarias a **Acabar** y **EsperarPor** (con la misma implementación que antes) para lograr la sincronización adecuada. Se incluye aquí una plantilla:

```

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

```

En este ejercicio contamos con dos alternativas: una que es más larga pero más general, y una más corta pero específica para este problema.

Matriz de esperas 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 comparada. Tenemos que:

```

1  var espera : array[1..9,1..2] of integer := (
       (-1, -1), { P1 }
       (1, -1), { P2 }
       (2, -1), { P3 }
5   (1, -1), { P4 }
       (2, 4), { P5 }
       (3, 5), { P6 }
       (4, -1), { P7 }
       (5, 7), { P8 }
10  (6, 8) { P9 }
   ) ;
   Process P[ n : 1..9 ]
   begin
       for i := 1 to 2 do
15         if espera[n,i] <> -1 then { != -1 }
           EsperarPor( espera[n,i] ) ;

           S_n ;
           Acabar( n ) ;
20  end

```

Usando operaciones En esta alternativa, emplearemos operaciones para determinar a qué procesos esperar.

```

1  Process P[ n : 1..9 ]
   begin
       {Esperamos al proceso de arriba}
       if (n > 3) then
5         EsperarPor( n-3 ) ;

       {Esperamos al proceso de la izquierda}
       if (n mod 3) <> 1 then
           EsperarPor( n-1 ) ;

```

10

```

    S_n ;
    Acabar( n ) ;
end

```

Ejercicio 5.1.10. 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:

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

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

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

Uniando 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:

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

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:

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

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

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

Uniando 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:

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

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

Mediante el axioma de asignación, tenemos que:

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

- b) $\{i > j + 1\} \quad j = j + 1 \quad \{i > j\}$

Mediante el axioma de asignación, tenemos que:

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

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, $\{\text{verdad}\}$.

5. $\{\text{verdad}\} \quad i = 3; j = 2 * i \quad \{\}$

Como partimos de cualquier estado del programa y sólo se realizan asignaciones, es fácil intuir cuál será la poscondición:

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

Pasamos a demostrar el triple

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

Usando la regla de composición, nos será suficiente probar los triples:

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

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

$$\{\text{verdad}\} \equiv \{3 = 3\} \equiv \{i = 3\}_3^i \quad i = 3 \quad \{i = 3\}$$

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

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

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

$$\begin{aligned} \{i = 3 \wedge j = 6\}_{2*i}^j &\equiv \{i = 3 \wedge 2 * i = 6\} \equiv \{i = 3 \wedge 2 * 3 = 6\} \equiv \\ &\equiv \{i = 3 \wedge 6 = 6\} \equiv \{i = 3\} \end{aligned}$$

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:

$$\begin{aligned} c &= a + b \\ c' &= c/2 = \frac{a + b}{2} \end{aligned}$$

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:

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

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

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

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

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

Usando la regla de composición, tenemos demostrado nuestro triple.

Ejercicio 5.1.11. ¿Cuáles de los siguientes triples no son demostrables con la Lógica de Programas? (Considerando que $i, x, a \in \mathbb{Z}$)

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

El siguiente triple sabemos que es cierto:

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

Y como $\{i > -1\} \equiv \{i \geq 0\}$, el triple es cierto.

2. $\{x \geq 7\} \quad x = x + 3; \quad \{x \geq 9\}$

El siguiente triple sabemos que es cierto:

$$\{x \geq 7\} \quad x = x + 3 \quad \{x \geq 10\}$$

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

3. $\{i < 9\} \quad i = 2 * i + 1; \quad \{i \leq 20\}$

El siguiente triple sabemos que es cierto:

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

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

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

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

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

Ejercicio 5.1.12. Si el triple $\{P\}C\{Q\}$ es demostrable, indicar por qué los siguientes triples también lo son (o no se pueden demostrar y por qué):

1. $\{P\}C\{Q \vee P\}$

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

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

Tenemos que se debilita la poscondición.

2. $\{P \wedge D\}C\{Q\}$

Es demostrable, ya que $\{P \wedge D\} \rightarrow \{P\}$ y por la segunda regla de la consecuencia, tomando $R = P \wedge D$:

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

Tenemos que se fortalece la precondition.

3. $\{P \vee D\}C\{Q\}$

No es demostrable, porque se debilita la precondition.

4. $\{P\}C\{Q \vee D\}$

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

5. $\{P\}C\{Q \wedge P\}$

No podemos demostrarlo, ya que se fortalece la poscondición.

Ejercicio 5.1.13. 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 $\{P \wedge D\} \rightarrow \{P\}$, luego puede demostrarse por la segunda regla de la consecuencia (se fortalece la precondition).

2. $\{P \vee D\}C\{Q\}$

No puede demostrarse, porque se debilita la precondition.

3. $\{P\}C\{Q \vee D\}$

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

4. $\{P\}C\{Q \vee P\}$

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

Ejercicio 5.1.14. Dado el siguiente programa, obtener:

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

1. Valores finales de x e y . 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 5.1.15. Comprobar si la demostración del siguiente triple interfiere con los teoremas siguientes:

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

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

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

Comprobamos en primer lugar su interferencia con la precondition:

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

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

Comprobemos ahora su interferencia con la poscondición:

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

En este caso, $\{x \geq 1\} \not\vdash \{x \geq 3\}$, luego este triple no es demostrable y R interfiere con la poscondición del triple en cuestión.

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

Comprobamos en primer lugar su interferencia con la precondition:

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

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

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

Comprobamos en primer lugar su interferencia con la precondition:

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

No obstante, como $\{x \geq 5\} \not\vdash \{x \geq 7\}$, R interfiere con la precondition de este triple.

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

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 precondition:

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

Por la 1ª regla de la consecuencia, como $\{x \text{ es impar} \wedge x \geq 0\} \rightarrow \{x \text{ es impar}\}$, tenemos que es correcto y R no interfiere con la precondition.

Comprobamos ahora su interferencia con la poscondición:

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

Por la 1ª regla de la consecuencia, como $\{y \text{ es par} \wedge x \geq 0\} \rightarrow \{y \text{ es par}\}$, tenemos que es correcto y R no interfiere con la poscondición. Por tanto, R no interfiere con este triple.

Ejercicio 5.1.16. Dado el siguiente triple:

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

Demostrarlo utilizando la lógica de asertos para cada una de las tres instrucciones atómicas y después que se llega a la poscondición final $x = a + b + c$ utilizando para ello la regla *de la composición concurrente* de instrucciones atómicas.

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:

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

Mediante el axioma de asignación, tenemos que:

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

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

$$\begin{aligned} & \{x = a \vee x = a + b \vee x = a + c \vee x = a + b + c\}_{x+a}^x \equiv \\ \equiv & \{x + a = a \vee x + a = a + b \vee x + a = a + c \vee x + a = a + b + c\} \equiv \\ \equiv & \{x = 0 \vee x = b \vee x = c \vee x = b + c\} \end{aligned}$$

Por tanto, el triple en cuestión es cierto.

2. El correspondiente a la segunda instrucción atómica:

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

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

3. El correspondiente a la tercera instrucción atómica:

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

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:

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

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

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

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:

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

Ejercicio 5.1.17. El siguiente triple:

$$\begin{aligned}
& \{x = 0 \wedge y = 0 \wedge z = 0\} \\
& < x = z + a > \parallel < y = x + b > \\
& \{(x = a) \wedge (y = b \vee y = a + b) \wedge z = 0\}
\end{aligned}$$

- (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:

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

Veamos cuál ha de ser la precondition 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:

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

Por tanto, del Axioma de Asignación, tenemos que:

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

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:

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

Por tanto, del Axioma de Asignación, tenemos que:

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

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:

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

Veamos por tanto en qué queda cada uno de ellos:

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

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

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

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 $\{x = 0 \wedge y = b \wedge z = 0\} \not\vdash \{y = a + b \wedge z = 0\}$, 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:

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

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 $\{x = a \wedge y = 0 \wedge z = 0\} \rightarrow \{x = 0 \wedge z = 0\}$, y por tanto es cierto. Por tanto, $NI(Q_1, S_2)$ 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:

$$\begin{aligned} & \{x = 0 \wedge y = 0 \wedge z = 0\} \\ & < x = z + a > \parallel < y = x + b > \\ & \{x = a \wedge y = a + b \wedge z = 0\} \end{aligned}$$

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

Ejercicio 5.1.18. Suponer que $\{suma > 1\} \text{ suma} = \text{suma} + 4 \{suma > 5\}$ es demostrable, entonces: ¿cuál de los siguientes triples es también demostrable? (indicar por qué)

1. $\{suma > 2\} \text{ suma} = \text{suma} + 4 \{suma > 5\}$.
Es demostrable, ya que $\{suma > 2\} \rightarrow \{suma > 1\}$ y podemos aplicar la segunda regla de la consecuencia.
2. $\{suma \geq 1\} \text{ suma} = \text{suma} + 4 \{suma > 5\}$.
No es demostrable, ya que debilita la precondition.
3. $\{suma > 0\} \text{ suma} = \text{suma} + 4 \{suma > 5\}$.
No es demostrable, ya que debilita la precondition.
4. $\{suma > 1\} \text{ suma} = \text{suma} + 4 \{suma > 6\}$.
No es demostrable, ya que fortalece la poscondición.

Ejercicio 5.1.19. Suponer que $\{x < y\} C_1 \{u < v\}$ es demostrable, entonces: ¿cuáles de los siguientes triples son también demostrables? (indicar por qué)

1. $\{x \leq y\} C_1 \{u < v\}$.
No es demostrable, ya que debilita la precondition.
2. $\{x \leq y - 2\} C_1 \{u < v\}$.
Es demostrable, ya que $\{x \leq y - 2\} \rightarrow \{x + 2 \leq y\} \rightarrow \{x < y\}$, y mediante la segunda regla de la consecuencia se tiene que es cierto.
3. $\{x \leq y\} C_1 \{u \leq v\}$.
El triple $\{x < y\} C_1 \{u \leq v\}$ sí que es demostrable ya que relaja la poscondición, pero el triple que se nos dice no es demostrable, ya que también relaja la precondition. 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 5.1.20. 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. $\langle x=x+y \rangle$
2. $\langle y=x*y \rangle$
3. $\langle x=x-y \rangle$

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 5.1.21. 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:

$$S_1 = S_2 = \langle x = x + a \rangle$$

Veamos cuál ha de ser la precondition 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:

$$\{x = 0 \vee x = a\} \langle x = x + a \rangle \{x = a \vee x = 2a\}$$

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

$$NI(P_1, S_2) \equiv \{x = 0 \vee x = a\} \langle x = x + a \rangle \{x = 0 \vee x = a\}$$

$$NI(Q_1, S_2) \equiv \{x = a\} \langle x = x + a \rangle \{x = a \vee x = 2a\}$$

El primer triple no es cierto, ya que por el Axioma de Asignación, tenemos que el triple correcto es:

$$\{x = -a \vee x = 0\} \langle x = x + a \rangle \{x = 0 \vee x = a\}$$

Como estamos debilitando la precondition, este triple no es cierto. Por tanto, la respuesta correcta es la **c)**, ya que al aplicar directamente la regla de inferencia de la *composición concurrente* utilizo unas condiciones (pre y post-condiciones) demasiado débiles.

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 preconditiones 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 5.1.22. 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:

$$\begin{aligned}
 &\{x = c_1 \wedge y = c_2\} \\
 &\quad x = x + y \\
 &\{x = c_1 + c_2 \wedge y = c_2\} \\
 &\quad y = x * y \\
 &\{x = c_1 + c_2 \wedge y = (c_1 + c_2) \cdot c_2\} \\
 &\quad x = x - y \\
 &\{x = (c_1 + c_2) - (c_1 + c_2) \cdot c_2 = (c_1 + c_2) \cdot (1 - c_2) \wedge y = (c_1 + c_2) \cdot c_2\}
 \end{aligned}$$

Ejercicio 5.1.23. Demostrar que el siguiente triple es cierto:

$$\begin{aligned}
 &\{x = 0\} \\
 &\quad \text{cobegin} \\
 &\quad \quad < x = x + 1 > \parallel < x = x + 2 > \parallel < x = x + 4 > \\
 &\quad \quad \text{coend} \\
 &\{x = 7\}
 \end{aligned}$$

Notamos cada instrucción atómica de la siguiente forma:

$$\begin{aligned}
 S_1 &= < x = x + 1 > \\
 S_2 &= < x = x + 2 > \\
 S_3 &= < x = x + 4 >
 \end{aligned}$$

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

$$\begin{aligned}
 P_1 &= x = 0 \vee x = 2 \vee x = 4 \vee x = 6 \\
 P_2 &= x = 0 \vee x = 1 \vee x = 4 \vee x = 5 \\
 P_3 &= x = 0 \vee x = 1 \vee x = 2 \vee x = 3
 \end{aligned}$$

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

$$\begin{aligned}
 Q_1 &= x = 1 \vee x = 3 \vee x = 5 \vee x = 7 \\
 Q_2 &= x = 2 \vee x = 3 \vee x = 6 \vee x = 7 \\
 Q_3 &= x = 4 \vee x = 5 \vee x = 6 \vee x = 7
 \end{aligned}$$

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:

$$\begin{array}{c} \{x = 0\} \\ \text{cobegin} \\ < x = x + 1 > \parallel < x = x + 2 > \parallel < x = x + 4 > \\ \text{coend} \\ \{x = 7\} \end{array}$$

Ejercicio 5.1.24. Dada la siguiente construcción de composición concurrente P:

$$\begin{array}{c} \text{cobegin} \\ < x = x - 1 >; < x = x + 1 >; \parallel < y = y - 1 >; < y = y + 1 >; \\ \text{coend} \end{array}$$

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)

$$\begin{array}{l} \{x = y\} \ x = x - 1; x = x + 1; \ \{x = y\} \\ \{x = y\} \ y = y - 1; y = y + 1; \ \{x = y\} \end{array}$$

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

$$\begin{array}{l} \{x = y\} \ x = x - 1 \ \{x + 1 = y\} \\ \{x + 1 = y\} \ x = x + 1 \ \{x = y\} \end{array}$$

Usando la regla de la composición, se tiene.

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

$$\begin{aligned} &\{y = x\} \ y = y - 1 \ \{y + 1 = x\} \\ &\{y + 1 = x\} \ y = y + 1 \ \{y = x\} \end{aligned}$$

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:

$$\begin{aligned} &\{x = y\} \\ &\text{cobegin} \\ &\quad < x = x - 1 >; < x = x + 1 >; || < y = y - 1 >; < y = y + 1 >; \\ &\quad \text{coend} \\ &\{x = y\} \end{aligned}$$

Ejercicio 5.1.25. 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:

$$\begin{aligned} &\{V\} \ i = 2 * i \ \{i = 2 * i\} \\ &\{i > 2\} \ i = 2 * i \ \{i > 2\} \end{aligned}$$

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:

$$\{i > 2\} \equiv \{V \wedge i > 2\} \ i = 2 * i \ \{i > 2 \wedge i = 2 * i\} \equiv \{i > 4\}$$

Ejercicio 5.1.26. Se dan los siguientes triples de Hoare:

$$\begin{aligned} &\{j > 1\} \ i = i + 2; \ j = j + 3; \ \{j > 4\} \\ &\{i > 2\} \ i = i + 2; \ j = j + 3; \ \{i > 4\} \end{aligned}$$

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 5.1.27. 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 $C = \langle a = a + b; b = a - 2b \rangle$. Buscamos entonces demostrar los siguientes triples:

$$\begin{aligned} & \{a = A \wedge b = B\} a = a + b \{a = A + B \wedge b = B\} \\ & \{a = A + B \wedge b = B\} b = a - 2b \{a = A + B \wedge b = A - B\} \end{aligned}$$

Demostramos cada uno por separado:

1. Usando el axioma de asignación:

$$\begin{aligned} \{a = A + B \wedge b = B\}_{a+b}^a &\equiv \{a + b = A + B \wedge b = B\} \equiv \\ &\equiv \{a = A \wedge b = B\} a = a + b \{a = A + B \wedge b = B\} \end{aligned}$$

2. Usando el axioma de asignación:

$$\begin{aligned} \{a = A + B \wedge b = A - B\}_{a-2b}^b &\equiv \{a = A + B \wedge a - 2b = A - B\} \equiv \\ &\equiv \{a = A + B \wedge A + B - 2b = A - B\} \equiv \\ &\equiv \{a = A + B \wedge b = B\} b = a - 2b \{a = A + B \wedge b = A - B\} \end{aligned}$$

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

Ejercicio 5.1.28. Demostrar que la siguiente sentencia tiene la poscondición $\{x \geq 0, x^2 \geq a^2\}$:

`if a > 0 then x = a else x = -a`

Es decir, probar el triple:

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

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

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

Luego bastará con probar los triples

$$\begin{aligned} \{a > 0\} &\equiv \{V \wedge a > 0\} x = a \{x \geq 0 \wedge x^2 \geq a^2\} \\ \{a \leq 0\} &\equiv \{V \wedge a \leq 0\} x = -a \{x \geq 0 \wedge x^2 \geq a^2\} \end{aligned}$$

1. Usando el axioma de asignación:

$$\{a \geq 0\} \equiv \{a \geq 0 \wedge a^2 \geq a^2\} \equiv \{x \geq 0 \wedge x^2 \geq a^2\}_a^x x = a \{x \geq 0 \wedge x^2 \geq a^2\}$$

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

2. Usando el axioma de asignación:

$$\{a \leq 0\} \equiv \{-a \geq 0 \wedge a^2 \geq a^2\} \equiv \{x \geq 0 \wedge x^2 \geq a^2\}_{-a}^x x = -a \{x \geq 0 \wedge x^2 \geq a^2\}$$

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

Ejercicio 5.1.29. El siguiente fragmento de código tiene $\{P\} \equiv \left\{sum = \frac{j(j-1)}{2}\right\}$ como precondition y poscondition. Demostrar que es verdadero:

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

Queremos demostrar el triple:

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

Para ello, será suficiente con demostrar los triples

$$\begin{aligned} &\left\{sum = \frac{j(j-1)}{2}\right\} \quad sum = sum + j; \quad \left\{sum = \frac{(j+1)j}{2}\right\} \\ &\left\{sum = \frac{(j+1)j}{2}\right\} \quad j = j + 1; \quad \left\{sum = \frac{j(j-1)}{2}\right\} \end{aligned}$$

y aplicar la regla de composición.

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

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

Usando la definición de Sustitución Textual, tenemos que:

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

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

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

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

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

Por lo que el triple del enunciado es cierto.

Ejercicio 5.1.30. 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 precondition del segundo triple.

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

Usando esa precondition, 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:

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

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

Ejercicio 5.1.31. ¿Por qué en la regla del **while** B, la condición B debe ser verdadera al comienzo del bucle?

Ejercicio 5.1.32. 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 5.1.33. 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:

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

Identificando términos, sean:

$$I \equiv \text{sum} = \frac{j(j-1)}{2}$$

$$B \equiv j \neq c$$

$$S \equiv \text{sum} = \text{sum} + j; \ j = j + 1$$

Luego tendremos que probar que se cumple el triple

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

Para ello, será suficiente con demostrar los triples

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

y aplicar la regla de composición.

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

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

$$\begin{aligned} & \left\{ sum = \frac{(j+1)j}{2} \wedge j \neq c \right\} \xrightarrow[sum+j]{sum} \equiv \left\{ sum + j = \frac{(j+1)j}{2} \wedge j \neq c \right\} \equiv \\ & \equiv \left\{ sum = \frac{(j+1)j}{2} - j \wedge j \neq c \right\} \equiv \left\{ sum = \frac{j(j-1)}{2} \wedge j \neq c \right\} \end{aligned}$$

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

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

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

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:

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

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

Ejercicio 5.1.34. Demostrar la corrección del siguiente triple:

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

Distinguiamos en función de los valores de i y j :

- Si $i = j$, entonces la poscondición queda $\{V\}$, luego el siguiente triple es cierto:

$$\{V\} \ a[i] = a[i] + a[i]; \ \{a[i] \geq a[i]\} \equiv \{V\}$$

Por tanto, como $\{a[i] \geq 0\} \subset \{V\}$, se tiene que $\{a[i] \geq 0\} \rightarrow \{V\}$, por lo que el triple del enunciado es cierto.

- Si $i \neq j$, entonces basta aplicar el axioma de la asignación:

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

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

Ejercicio 5.1.35. Verificar el siguiente segmento de programa:

```

    {n ≥ 0}
    i = 1;
    while i ≤ n do begin
        a[i] = b[i];
        i = i + 1;
    end
    { ⋀_{i=1}^n (a[i] = b[i]) }

```

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:

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

- Usando el axioma de asignación, tenemos que:

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

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

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

Posteriormente, hemos de demostrar que $\{I \wedge B\} S \{I\}$ con $\{B\} \equiv \{i \leq n\}$ 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:

$$\begin{aligned} \{I \wedge B\} &\equiv \left\{ \bigwedge_{j=1}^{i-1} (a[j] = b[j]) \wedge i \leq n \right\} \\ &\quad a[i] = b[i]; \\ &\quad \left\{ \bigwedge_{j=1}^i (a[j] = b[j]) \wedge i \leq n \right\} \\ &\quad i = i + 1; \\ &\quad \left\{ \bigwedge_{j=1}^{i-1} (a[j] = b[j]) \wedge i \leq n + 1 \right\} \rightarrow \{I\} \end{aligned}$$

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

$$\begin{aligned} \{I\} &\equiv \left\{ \bigwedge_{j=1}^{i-1} (a[j] = b[j]) \right\} \\ &\quad \text{while } i \leq n \text{ do begin} \\ &\quad \quad a[i] = b[i]; \\ &\quad \quad i = i + 1; \\ &\quad \quad \text{end} \\ \{I \wedge \neg B\} &\equiv \left\{ \bigwedge_{j=1}^{i-1} (a[j] = b[j]) \wedge i > n \right\} \equiv \left\{ \bigwedge_{j=1}^{i-1} (a[j] = b[j]) \wedge i - 1 \geq n \right\} \rightarrow \left\{ \bigwedge_{j=1}^n (a[j] = b[j]) \right\} \end{aligned}$$

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

$$\begin{aligned} &\{n \geq 0\} \\ &\quad i = 1; \\ &\quad \{I\} \\ &\quad \text{while } i \leq n \text{ do begin} \\ &\quad \quad a[i] = b[i]; \\ &\quad \quad i = i + 1; \\ &\quad \quad \text{end} \\ &\quad \left\{ \bigwedge_{i=1}^n (a[i] = b[i]) \right\} \end{aligned}$$

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

Ejercicio 5.1.36. 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:

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

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:

$$\begin{aligned} \{I \wedge B\} &\equiv \left\{ sum = \sum_{j=1}^{i-1} j! \wedge f = i! \wedge i \neq n+1 \right\} \\ &\quad sum = sum + f; \\ \left\{ sum = \left(\sum_{j=1}^{i-1} j! \right) + i! \wedge f = i! \wedge i \neq n+1 \right\} &\equiv \left\{ sum = \sum_{j=1}^i j! \wedge f = i! \wedge i \neq n+1 \right\} \\ &\quad i = i + 1; \\ \left\{ sum = \sum_{j=1}^{i-1} j! \wedge f = (i-1)! \wedge i \neq n+2 \right\} & \\ &\quad f = f * i; \\ \left\{ sum = \sum_{j=1}^{i-1} j! \wedge f = i \cdot (i-1)! \wedge i \neq n+2 \right\} &\equiv \left\{ sum = \sum_{j=1}^{i-1} j! \wedge f = i! \wedge i \neq n+2 \right\} \rightarrow \{I\} \end{aligned}$$

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

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

$$\begin{array}{l} \text{while } i \neq n+1 \text{ do begin} \\ \quad \text{sum} = \text{sum} + f; \\ \quad i = i + 1; \\ \quad f = f * i; \\ \text{end} \end{array}$$

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

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

Ejercicio 5.1.37. Hallar la precondition $\{P\}$ que hace que el siguiente triple sea correcto:

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

Para ello, basta aplicar el axioma de asignación:

$$\begin{array}{l} \{j \leq i \wedge k < i \wedge a[i] + a[j-1] + a[k] > b\} \xrightarrow{a[i] = 2 * b} \{j \leq i \wedge k < i \wedge 2 \cdot b + a[j-1] + a[k] > b\} \\ \equiv \{j \leq i \wedge k < i \wedge b + a[j-1] + a[k] > 0\} \end{array}$$

Usando la definición de Sustitución Textual, tenemos que:

$$\begin{array}{l} \{j \leq i \wedge k < i \wedge a[i] + a[j-1] + a[k] > b\} \xrightarrow{a[i] = 2 \cdot b} \{j \leq i \wedge k < i \wedge 2 \cdot b + a[j-1] + a[k] > b\} \\ \equiv \{j \leq i \wedge k < i \wedge b + a[j-1] + a[k] > 0\} \end{array}$$

Luego estamos buscando la precondition:

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

Ejercicio 5.1.38. Demostrar que para $n > 0$ el siguiente fragmento de programa termina.

```

1  i = 1; f = 1;
   while i <> n do begin
       i = i + 1;
       f = f * i;
5  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:

$$\{1, 2, \dots, n\}$$

Por tanto, nada impide que se llegue a $i = n$, y por tanto el bucle termina.

Ejercicio 5.1.39. Hallar la precondition 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 5.1.40. 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 $\{i > j\}$:

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

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 $\{i = 3 \wedge j = 6\}$:

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

Ejercicio 5.1.41. Para cada uno de los siguientes fragmentos de código, obtener las preconditiones apropiadas.

1. $i = 3 * k; \{i > 6\}$.

Aplicando el axioma de asignación:

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

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

2. $a = b * c; \{a = 1\}$.

Aplicando el axioma de asignación:

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

La precondition 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 precondition de la segunda instrucción debe ser $b \neq 0$. Es decir:

$$\{b \neq 0\} \quad a = a/b; \{V\}$$

Aplicando ahora el axioma de sustitución, tenemos que el siguiente triple es cierto:

$$\{c \neq 2\} \equiv \{c - 2 \neq 0\} \equiv \{b \neq 0\}_{c-2}^b \quad b = c - 2; \{b \neq 0\}$$

Por tanto, la precondition es $\{c \neq 2\}$.

Ejercicio 5.1.42. Obtener la poscondición más fuerte posible del siguiente código. Indicar todas las reglas usadas.

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

Respecto de la primera instrucción, tenemos por el axioma de asignación que:

$$\{y > 0\} \quad xa = x + y; \quad \{xa > x, y > 0\}$$

Para la segunda instrucción, por el axioma de asignación, tenemos que:

$$\{xa > x, y > 0\} \quad xb = x - y; \quad \{xb < xa - y, y > 0\}$$

Por tanto, por la regla de composición, tenemos que:

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

Aunque no se pide, como poscondición más débil, podemos obtener que:

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

Ejercicio 5.1.43. Verificar el siguiente código, indicando todas las reglas usadas.

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

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

$$\{x \leq 0\} \equiv \{-x \geq 0\} \equiv \{x \geq 0\}_{-x}^x \ x = -x; \{x \geq 0\}$$

Y como $\{x < 0\} \rightarrow \{x \leq 0\}$, tenemos que el triple $\{x < 0\} \ x = -x; \{x \geq 0\}$ es cierto, por fortalecer la precondition.

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

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

Ejercicio 5.1.44. Verificar el siguiente segmento de programa:

```

max = a[1]; i = 1;
while i <> n + 1 do begin
  if a[i] ≥ max then max = a[i];
  i = i + 1;
end
{
  ⋀_{i=1}^n (max ≥ a[i])
}
```

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 usamos¹ es el siguiente:

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

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

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

Lo cual es cierto, ya que:

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

Posteriormente, pasaremos a demostrar el triple $\{I \wedge B\} \ S \ \{I\}$ con $\{B\} \equiv \{i \neq n+1\}$

¹Para buscarlo, hemos de pensar en una regla que se mantenga iteración tras iteración.

y S el cuerpo del bucle para poder aplicar la regla de iteración:

$$\begin{aligned}
 \{I \wedge B\} &\equiv \left\{ \bigwedge_{j=1}^{i-1} (max \geq a[j]) \wedge i \neq n+1 \right\} \\
 &\quad \text{if } a[i] \geq max \text{ then } max = a[i]; \\
 &\quad \left\{ \bigwedge_{j=1}^i (max \geq a[j]) \wedge i \neq n+1 \right\} \\
 &\quad i = i + 1; \\
 &\left\{ \bigwedge_{j=1}^{i-1} (max \geq a[j]) \wedge i \neq n+2 \right\} \rightarrow \{I\}
 \end{aligned}$$

Donde hemos usando que el siguiente triple es cierto:

$$\begin{aligned}
 &\left\{ \bigwedge_{j=1}^{i-1} (max \geq a[j]) \wedge i \neq n+1 \right\} \\
 &\text{if } a[i] \geq max \text{ then } max = a[i]; \\
 &\left\{ \bigwedge_{j=1}^i (max \geq a[j]) \wedge i \neq n+1 \right\}
 \end{aligned}$$

Para comprobar su veracidad, hemos de ver que:

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

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:

$$\begin{aligned}
 \{I\} &\equiv \left\{ \bigwedge_{j=1}^{i-1} (max \geq a[j]) \right\} \\
 &\quad \text{while } i <> n+1 \text{ do begin} \\
 &\quad \text{if } a[i] \geq max \text{ then } max = a[i]; \\
 &\quad \quad i = i + 1; \\
 &\quad \text{end} \\
 \{I \wedge \neg B\} &\equiv \left\{ \bigwedge_{j=1}^{i-1} (max \geq a[j]) \wedge i = n+1 \right\} \equiv \left\{ \bigwedge_{j=1}^n (max \geq a[j]) \right\}
 \end{aligned}$$

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 5.1.45. Demostrar la corrección parcial del siguiente código:

```

    max = a[1]; i = 1;
    while i < n do begin
        i = i + 1;
        if a[i] ≥ max then max = a[i];
    end
    {
        ⋀i=1n (max ≥ a[i]), ⋁j=1n (max = a[j])
    }
```

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:

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

En primer lugar, hemos de ver que el invariante es cierto al inicio del programa:

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

Esto sabemos que es cierto, ya que:

$$\begin{aligned} & \{V\} \text{ max} = a[1]; i = 1; \{\text{max} = a[1] \wedge i = 1\} \\ & \{\text{max} = a[1] \wedge i = 1\} \rightarrow \{\text{max} \geq a[1] \wedge \text{max} = a[1] \wedge i = 1\} \rightarrow \{I\} \end{aligned}$$

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:

$$\begin{aligned} \{I \wedge B\} & \equiv \left\{ \bigwedge_{j=1}^i (max \geq a[j]) \wedge \bigvee_{j=1}^i (max = a[j]) \wedge i < n \right\} \\ & \quad i = i + 1; \\ & \quad \left\{ \bigwedge_{j=1}^{i-1} (max \geq a[j]) \wedge \bigvee_{j=1}^{i-1} (max = a[j]) \wedge i < n + 1 \right\} \\ & \quad \text{if } a[i] \geq max \text{ then } max = a[i]; \\ & \quad \left\{ \bigwedge_{j=1}^i (max \geq a[j]) \wedge \bigvee_{j=1}^i (max = a[j]) \wedge i < n + 1 \right\} \rightarrow \{I\} \end{aligned}$$

Donde hemos usado que el siguiente triple es cierto:

$$\begin{aligned} & \left\{ \bigwedge_{j=1}^{i-1} (max \geq a[j]) \wedge \bigvee_{j=1}^{i-1} (max = a[j]) \wedge i < n + 1 \right\} \\ & \quad \text{if } a[i] \geq max \text{ then } max = a[i]; \\ & \quad \left\{ \bigwedge_{j=1}^i (max \geq a[j]) \wedge \bigvee_{j=1}^i (max = a[j]) \wedge i < n + 1 \right\} \end{aligned}$$

Para comprobar su veracidad, hemos de ver que:

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

$$\left\{ \bigwedge_{j=1}^i (max \geq a[j]) \wedge \bigvee_{j=1}^i (max = a[j]) \wedge i < n + 1 \right\}$$

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

$$\left\{ \bigwedge_{j=1}^i (max \geq a[j]) \wedge \bigvee_{j=1}^i (max = a[j]) \wedge i < n + 1 \right\}$$

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:

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

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

$$i = i + 1;$$

$$\text{if } a[i] \geq max \text{ then } max = a[i];$$

$$\text{end}$$

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

$$\rightarrow \left\{ \bigwedge_{j=1}^i (max \geq a[j]) \wedge \bigvee_{j=1}^i (max = a[j]) \wedge i = n \right\} \equiv \left\{ \bigwedge_{j=1}^n (max \geq a[j]) \wedge \bigvee_{j=1}^n (max = a[j]) \right\}$$

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 5.1.46. Demostrar la corrección parcial del siguiente código:

```

i = 0; j = n;
while i < n do begin
    i = i + 1;
    j = j - 1;
    a[i] = b[j]
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 global 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:

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

donde hemos usado que

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

que puede probarse mediante composición de los triples

$$\begin{aligned} &\{j = n-i\} \ i = i + 1; \ \{j = n-i+1\} \\ &\{j = n-i+1\} \ j = j - 1; \ \{j = n-i\} \end{aligned}$$

que sabemos que son ciertos por el axioma de asignación.

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

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

$$\begin{array}{l} \text{while } i < n \text{ do begin} \\ \quad i = i + 1; \\ \quad j = j - 1; \\ \quad a[i] = b[j] \\ \text{end} \end{array}$$

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

Como hemos probado inicialmente que $\{V\} \ i = 0; \ j = n; \ \{I\}$, por la regla de la composición, hemos demostrado la corrección del programa.

Ejercicio 5.1.47. 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:

$$\begin{aligned}
\{I\} &\equiv \left\{ \bigwedge_{k=1}^{i-1} (a[k] = s) \wedge s = \sum_{k=1}^{i-1} A[k] \right\} \\
&\quad \textbf{while } i \leq n \textbf{ do begin} \\
&\quad \left\{ \bigwedge_{k=1}^{i-1} (a[k] = s) \wedge s = \sum_{k=1}^{i-1} A[k] \wedge i \leq n \right\} \\
&\quad \quad s = s + a[i]; \\
&\quad \left\{ \bigwedge_{k=1}^{i-1} (a[k] = s) \wedge s = \sum_{k=1}^{i-1} A[k] + A[i] \wedge i \leq n \right\} \equiv \\
&\quad \equiv \left\{ \bigwedge_{k=1}^{i-1} (a[k] = s) \wedge s = \sum_{k=1}^i A[k] \wedge i \leq n \right\} \\
&\quad \quad a[i] = s; \\
&\quad \left\{ \bigwedge_{k=1}^i (a[k] = s) \wedge s = \sum_{k=1}^i A[k] \wedge i \leq n \right\} \\
&\quad \quad i = i + 1; \\
&\quad \left\{ \bigwedge_{k=1}^{i-1} (a[k] = s) \wedge s = \sum_{k=1}^{i-1} A[k] \wedge i - 1 \leq n \right\} \rightarrow \left\{ \bigwedge_{k=1}^{i-1} (a[k] = s) \wedge s = \sum_{k=1}^{i-1} A[k] \right\} \equiv \{I\} \\
&\quad \textbf{end} \\
&\quad \left\{ \bigwedge_{k=1}^{i-1} (a[k] = s) \wedge s = \sum_{k=1}^{i-1} A[k] \wedge i > n \right\}
\end{aligned}$$

Donde en el último paso hemos aplicado la regla de la iteración: como $\{I \wedge B\} C \{I\}$ siendo $B = i \leq n$ y C el cuerpo del bucle, entonces se tiene que:

$$\{I\} \textbf{ while } B \textbf{ do } C \textbf{ end } \{I \wedge \neg B\}$$

Finalmente, tendremos que:

$$\left\{ \bigwedge_{k=1}^n (a[k] = s) \wedge s = \sum_{k=1}^n A[k] \right\}$$

Ejercicio 5.1.48. Dados $i, n \geq 0$, $i \leq n$, demostrar que el siguiente segmento de programa 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:

$$\{I\} \equiv \{fact = k! \wedge afact = (\min\{i, k\})! \wedge bfact = (\min\{n-i, k\})!\}$$

En primer lugar, hemos de ver que el invariante es cierto al inicio del programa:

$$\{0 \leq i \leq n\} \quad k = 0; \quad fact = 1; \quad \{I\}$$

Lo cual es directamente cierto². Posteriormente, hemos de demostrar el triple dado por $\{I \wedge B\} \quad S \quad \{I\}$ con $\{B\} \equiv \{k \neq n\}$ y S el cuerpo del bucle para poder aplicar la regla de iteración:

$$\begin{aligned}
\{I \wedge B\} &\equiv \{fact = k! \wedge afact = (\min\{i, k\})! \wedge bfact = (\min\{n-i, k\})! \wedge k \neq n\} \\
&\quad k = k + 1; \\
\{fact = (k-1)! \wedge afact = (\min\{i, k-1\})! \wedge bfact = (\min\{n-i, k-1\})! \wedge k \neq n+1\} \\
&\quad fact = fact * k; \\
\{fact = k! \wedge afact = (\min\{i, k-1\})! \wedge bfact = (\min\{n-i, k-1\})! \wedge k \neq n+1\} \\
&\quad \text{if } k \leq i \text{ then } afact = fact; \\
&\quad \{fact = k! \wedge afact = (\min\{i, k\})! \wedge bfact = (\min\{n-i, k-1\})! \wedge k \neq n+1\} \\
&\quad \text{if } k \leq n-i \text{ then } bfact = fact; \\
&\quad \{fact = k! \wedge afact = (\min\{i, k\})! \wedge bfact = (\min\{n-i, k\})! \wedge k \neq n+1\} \rightarrow \{I\}
\end{aligned}$$

donde hemos empleado dos veces la regla del **if**, veámoslo:

- En primer lugar, hemos de demostrar el siguiente triple:

$$\begin{aligned}
&\{fact = k! \wedge afact = (\min\{i, k-1\})! \wedge bfact = (\min\{n-i, k-1\})! \wedge k \neq n+1\} \\
&\quad \text{if } k \leq i \text{ then } afact = fact; \\
&\{fact = k! \wedge afact = (\min\{i, k\})! \wedge bfact = (\min\{n-i, k-1\})! \wedge k \neq n+1\}
\end{aligned}$$

Para demostrarlo, hemos de demostrar los dos siguientes triples:

- En primer lugar, hemos de demostrar que:

$$\begin{aligned}
&\{fact = k! \wedge afact = (\min\{i, k-1\})! \wedge bfact = (\min\{n-i, k-1\})! \wedge k \neq n+1 \wedge k \leq i\} \\
&\quad afact = fact; \\
&\{fact = k! \wedge afact = (\min\{i, k\})! \wedge bfact = (\min\{n-i, k-1\})! \wedge k \neq n+1\}
\end{aligned}$$

²Notemos que, para demostrar esto, hemos de suponer valores iniciales de **afact** y **bfact**, algo que no supone problema alguno.

Usando la segunda regla de la consecuencia, la sustitución textual y el axioma de asignación, tenemos que:

$$\begin{aligned}
& \{fact = k! \wedge a fact = (\text{mín}\{i, k-1\})! \wedge b fact = (\text{mín}\{n-i, k-1\})! \wedge k \neq n+1 \wedge k \leq i\} \rightarrow \\
& \rightarrow \{fact = k! \wedge b fact = (\text{mín}\{n-i, k-1\})! \wedge k \neq n+1 \wedge k \leq i\} \equiv \\
& \equiv \{fact = k! \wedge k! = (\text{mín}\{i, k\})! \wedge b fact = (\text{mín}\{n-i, k-1\})! \wedge k \neq n+1\} \equiv \\
& \equiv \{fact = k! \wedge fact = (\text{mín}\{i, k\})! \wedge b fact = (\text{mín}\{n-i, k-1\})! \wedge k \neq n+1\} \equiv \\
& \equiv \{fact = k! \wedge a fact = (\text{mín}\{i, k\})! \wedge b fact = (\text{mín}\{n-i, k-1\})! \wedge k \neq n+1\}_{fact}^{a fact} \\
& \quad a fact = fact; \\
& \{fact = k! \wedge a fact = (\text{mín}\{i, k\})! \wedge b fact = (\text{mín}\{n-i, k-1\})! \wedge k \neq n+1\}
\end{aligned}$$

- En segundo lugar, hemos de demostrar que:

$$\begin{aligned}
& \{fact = k! \wedge a fact = (\text{mín}\{i, k-1\})! \wedge b fact = (\text{mín}\{n-i, k-1\})! \wedge k \neq n+1 \wedge k > i\} \\
& \quad null \\
& \{fact = k! \wedge a fact = (\text{mín}\{i, k\})! \wedge b fact = (\text{mín}\{n-i, k-1\})! \wedge k \neq n+1\}
\end{aligned}$$

Teniendo en cuenta que $k > i \implies k-1 \geq i$, 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:

$$\begin{aligned}
& \{fact = k! \wedge a fact = (\text{mín}\{i, k\})! \wedge b fact = (\text{mín}\{n-i, k-1\})! \wedge k \neq n+1\} \\
& \quad \text{if } k \leq n-i \text{ then } b fact = fact; \\
& \{fact = k! \wedge a fact = (\text{mín}\{i, k\})! \wedge b fact = (\text{mín}\{n-i, k\})! \wedge k \neq n+1\}
\end{aligned}$$

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:

$$\begin{aligned}
& \{I\} \equiv \{fact = k! \wedge a fact = (\text{mín}\{i, k\})! \wedge b fact = (\text{mín}\{n-i, k\})!\} \\
& \quad \text{while } k <> n \text{ do begin} \\
& \quad \quad k = k + 1; \\
& \quad \quad fact = fact * k; \\
& \quad \quad \text{if } k \leq i \text{ then } a fact = fact; \\
& \quad \quad \text{if } k \leq n-i \text{ then } b fact = fact; \\
& \quad \text{end} \\
& \{I \wedge \neg B\} \equiv \{fact = k! \wedge a fact = (\text{mín}\{i, k\})! \wedge b fact = (\text{mín}\{n-i, k\})! \wedge k = n\}
\end{aligned}$$

Usando la regla de la composición (ya que hemos probado que $\{V\} k = 0; fact =$

1; $\{I\}$), y usando que $i \leq n$, tenemos que:

$$\begin{aligned} & \{V\} \\ & k = 0; \text{ fact} = 1; \text{while } k <> n \text{ do begin} \\ & \quad k = k + 1; \\ & \quad \text{fact} = \text{fact} * k; \\ & \quad \text{if } k \leq i \text{ then } \text{afact} = \text{fact}; \\ & \quad \text{if } k \leq n - i \text{ then } \text{bfact} = \text{fact}; \\ & \quad \text{end} \\ & \{ \text{fact} = k! \wedge \text{afact} = i! \wedge \text{bfact} = (n - i)! \} \\ & \text{bcof} = \text{fact} / (\text{afact} * \text{bfact}); \\ & \left\{ \text{bcof} = \frac{n!}{i!(n - i)!} \right\} \end{aligned}$$

donde para demostrar el último triple hemos empleado la segunda regla de la consecuencia, la sustitución textual y el axioma de asignación.

Ejercicio 5.1.49. Demostrar la terminación del fragmento de programa dado en el problema 5.1.44 ¿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:

$$\{1, 2, 3, \dots\}$$

Por tanto, para demostrar que llega al caso $i = n + 1$ (llegando así a terminar), es necesario que $n \geq 0$.

5.2. Sincronización en Memoria Compartida

5.2.1. Exclusión mutua

Ejercicio 5.2.1. 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.

Un algoritmo para el cual solo pudiéramos demostrar las 4 condiciones de Dijkstra solo contaría con la propiedad de seguridad de alcanzabilidad (la 4ª condición) y de exclusión mutua, ya que:

- Para la propiedad de vivacidad, deberíamos demostrar que el algoritmo se encuentra libre de interbloqueos.
- Para la propiedad de equidad, deberíamos demostrar que cada proceso es capaz de llegar a la sección crítica en un tiempo máximo, con lo que el reparto de la sección crítica es justo para los procesos.

Ejercicio 5.2.2. 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 5.2.3. ¿Podría pensarse que una posible solución al problema de la exclusión mutua, sería el siguiente algoritmo (de la Figura 5.9) 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)

```

1  {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}

1  Process P0;
begin
while true do begin
    {Protocolo de entrada}

5      {indica que quiere entrar}
      b0 := true;
      {si el otro también}
      while b1 do begin
10         {cede temporalmente}
         b0 := false;
         {espera}
         while b1 do begin end
         {vuelve a cerrar el paso}
15         b0 := true;
      end

      {Sección crítica}
      {Protocolo de salida}
20      b0 := false;
      {Resto de sentencias}
end {while}
end

1  Process P1;
begin
while true do begin
    {Protocolo de entrada}

5      {indica que quiere entrar}
      b1 := true;
      {si el otro también}
      while b0 do begin
10         {cede temporalmente}
         b1 := false;
         {espera}
         while b0 do begin end
         {vuelve a cerrar el paso}
15         b1 := true;
      end

      {Sección crítica}
      {Protocolo de salida}
20      b1 := false;
      {Resto de sentencias}
end {while}
end

```

Figura 5.9: Código para el Ejercicio 5.2.3.

El código de la Figura 5.9 se corresponde con el algoritmo correspondiente a la cuarta etapa del método de refinamiento sucesivo de Dijkstra:

- (a) La exclusión mutua puede demostrarse: Suponiendo que nos interesa ver las condiciones bajo las cuales el proceso P0 entra en la sección crítica (para el proceso P1 el razonamiento es análogo), este podrá entrar en la sección crítica solo si `b1 = false`, que se dará siempre que:
- El proceso P1 no quiera entrar a la sección crítica (esto es, que este pasivo).
 - O bien que haya cambiado su clave a `false` en la instrucción de la línea 11, con lo que quedará bloqueado en el bucle más interno (porque `b0 = true`).

Como ninguna de estas situaciones es compatible con que el proceso P1 entre a la vez con P0 a la sección crítica, concluimos que es imposible que P0 y P1 entren a la vez en sección crítica.

- (b) La ausencia de interbloqueo no puede demostrarse:

Si suponemos que P0 y P1 tienen velocidades de ejecución idénticas y que comienzan con su ejecución en el mismo instante, podemos observar la traza de ejecución:

- Ambos cambian su clave a **true**.
- Ambos ven que la clave del otro está a **true**.
- Ambos cambian su clave a **false** (no ejecutan el bucle más interno).
- Ambos cambian su clave a **true**.
- Ambos ven que la clave del otro está a **true**.
- Ambos cambian su clave a **false** (no ejecutan el bucle más interno).
- Ambos cambian su clave a **true**.
- ...

Que lleva a un interbloqueo de ambos procesos, de forma que ninguno consiga al final entrar a la sección crítica.

Ahora podemos contestar a la pregunta inicial: aunque el algoritmo garantiza la propiedad de seguridad de la exclusión mutua, como hay una situación de interbloqueo, la bondad del algoritmo depende de la velocidad de ejecución de los procesos que ejecuten el programa, por lo que decimos que no es una buena solución al problema de la exclusión mutua.

Ejercicio 5.2.4. Al siguiente algoritmo (de la Figura 5.21) 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?

```

1  {variables compartidas y valores iniciales}
var c0 : integer := 1;
    c1 : integer := 1;
    turno : integer := 1;

```

```

1  process P0;
begin
while true do begin
    c0 := 0;
5   while turno <> 0 do begin
        while c1 = 0 do begin end
        turno := 0;
    end
10  {Sección crítica}
    c0 := 1;
    {Resto de sentencias}
end
end

```

```

1  process P1;
begin
while true do begin
    c1 := 0;
5   while turno <> 1 do begin
        while c0 = 0 do begin end
        turno := 1;
    end
10  {Sección crítica}
    c1 := 1;
    {Resto de sentencias}
end
end

```

Figura 5.10: Código para el Ejercicio 5.2.4.

No es correcta, ya que no se da la propiedad de seguridad de la exclusión mutua: supongamos que P0 y P1 ejecutan su código de forma que se da la traza de ejecución de la Tabla 5.2 (donde indicamos la línea de la instrucción que ejecuta cada proceso en cada instante).

Línea de P0	Línea de P1	c0	c1	turno
3	3	1	1	1
4	3	0	1	1
5	3	0	1	1
6	3	0	1	1
6	4	0	0	1
6	5	0	0	1
6	10	0	0	1
7	10	0	0	0
10	10	0	0	0

Tabla 5.2: Traza de ejecución que lleva a tener dos procesos en SC.

Ejercicio 5.2.5. Supongamos el algoritmo de exclusión mutua que expresamos a continuación. Tenemos los procesos: $0, 1, \dots, n-1$. 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 y solo si se cumplen las siguientes condiciones:

$$s[i] \neq s[i-1] \text{ para } i > 0$$

$$s[0] = s[n-1] \text{ para } i = 0$$

Tras ejecutar su sección crítica, el proceso i deberá hacer:

$$s[i] = s[i-1] \text{ para } i > 0$$

$$s[0] = (s[0] + 1) \text{ mód } 2 \text{ para } i = 0$$

Se pide:

- (i) Demostrar que el algoritmo anterior solo permite a un proceso acceder a la sección crítica en cualquier configuración (es decir, independientemente de los valores que pueden alcanzar las variables $s[i]$ durante la ejecución del protocolo).
- (ii) Demostrar también que cada uno de los procesos $p[i]$ conseguirá entrar en la sección crítica “infinitamente a menudo”.

Para probar los dos apartados, introducimos primero una notación que nos ayudará a realizar la demostración:

Definición 5.1. Sea P_k un proceso (con $k \in \{0, \dots, n-1\}$), diremos que es distinguido si:

$$s[i] = j \quad \forall i \in \{0, \dots, k-1\}$$

$$s[i] = (j+1) \text{ mód } 2 \quad \forall i \in \{k, \dots, n-1\}$$

Para cierto $j \in \{0, 1\}$.

De esta forma, con la configuración inicial ($s[i] = 0 \forall i \in \{1, \dots, n-1\}$), tenemos que el proceso P_0 y solo él es distinguido. Notemos que si dos procesos son distinguidos al mismo tiempo entonces son iguales, por lo que solo puede haber como máximo un proceso distinguido a la vez.

Proposición 5.1. *Si el proceso P_k con $k \in \{0, \dots, n-2\}$ es distinguido y pasa a ejecutar la sección crítica, tras modificar el valor $s[k]$, el proceso P_{i+1} será distinguido.*

Análogamente, si el proceso P_{n-1} es distinguido, tras ejecutar la sección crítica y modificar $s[n-1]$, el proceso P_0 será distinguido.

Demostración. Demostremos los dos enunciados:

- Supuesto que el proceso P_k con $k \in \{0, \dots, n-2\}$ es distinguido, entonces estaremos en un estado del programa de forma que:

$$\begin{aligned} s[i] &= j & \forall i \in \{1, \dots, k-1\} \\ s[i] &= (j+1) \text{ mód } 2 & \forall i \in \{k, \dots, n-1\} \end{aligned}$$

Para cierto $j \in \{0, 1\}$. Notemos que esta hipótesis no es incompatible con que P_k ejecute la sección crítica, ya que:

$$s[k] = (j+1) \text{ mód } 2 \neq j = s[k-1]$$

Tras su ejecución, el proceso realizará la instrucción:

Supuesto que $k = 0$:

$s[0] = (s[0]+1) \text{ mód } 2$, con lo que:

$$\begin{aligned} \left\{ \bigwedge_{i=0}^{n-1} s[i] = (j+1) \text{ mód } 2 \right\} & s[0] = (s[0]+1) \text{ mód } 2; \\ & \left\{ \left(\bigwedge_{i=1}^{n-1} s[i] = (j+1) \text{ mód } 2 \right) \wedge s[0] = j \right\} \end{aligned}$$

Por lo que ahora el proceso $P_{k+1} = P_1$ es distinguido.

Supuesto que $k \neq 0$:

$s[k] = s[k-1]$, por lo que en este caso:

$$\begin{aligned} \left\{ \left(\bigwedge_{i=0}^{k-1} s[i] = j \right) \wedge \left(\bigwedge_{i=k}^{n-1} s[i] = (j+1) \text{ mód } 2 \right) \right\} & s[k] = s[k-1]; \\ & \left\{ \left(\bigwedge_{i=0}^k s[i] = j \right) \wedge \left(\bigwedge_{i=k+1}^{n-1} s[i] = (j+1) \text{ mód } 2 \right) \right\} \end{aligned}$$

Por lo que ahora el proceso P_{k+1} es distinguido.

- Supongamos que el proceso P_{n-1} es distinguido, con lo que tendremos que $s[i] = j \forall i \in \{0, \dots, n-2\}$ para cierto $j \in \{0, 1\}$ y que $s[n-1] = (j+1) \text{ mód } 2$.

Notemos que la hipótesis de que P_{n-1} ejecuta la sección crítica es compatible con ser distinguido, ya que:

$$s[n-1] = (j+1) \pmod 2 \neq j = s[n-2]$$

Después de que P_{n-1} ejecute la sección crítica, realizará la asignación $s[n-1] = s[n-2]$, con lo que:

$$\left\{ \left(\bigwedge_{i=0}^{n-2} s[i] = j \right) \wedge s[n-1] = (j+1) \pmod 2 \right\} s[n-1] = s[n-2];$$

$$\left\{ \bigwedge_{i=0}^{n-1} s[i] = j \right\}$$

Y tenemos que P_0 es distinguido.

□

Notemos que en dicha proposición también hemos probado que si un proceso es distinguido, entonces cumplirá la condición para poder entrar a la sección crítica.

Ahora estamos en las condiciones de demostrar los dos apartados:

(i) Demostraremos que el algoritmo de entrada a la sección crítica solo permite el paso a un proceso:

- Inicialmente, como $s[i] = 0 \forall i \in \{0, \dots, n-1\}$, entonces el proceso P_0 es distinguido, con lo que podrá entrar en la sección crítica. Veamos que ningún otro podrá entrar a la vez, ya que si $k \in \{1, \dots, n-1\}$, P_k no podrá entrar en sección crítica, ya que:

$$s[k] = 0 = 0 = s[k-1]$$

Con lo que inicialmente P_0 es el único proceso que puede entrar en la sección crítica.

- Supuesto que P_k con $k \in \{0, \dots, n-1\}$ es un proceso distinguido, entonces tendremos por definición que:

$$\begin{aligned} s[i] &= j & \forall i \in \{0, \dots, k-1\} \\ s[i] &= (j+1) \pmod 2 & \forall i \in \{k, \dots, n-1\} \end{aligned}$$

Con lo que podrá entrar en la sección crítica. Sea $h \in \{0, \dots, n-1\} \setminus \{k\}$:

- Si $h = 0$ (entonces teníamos $k > 0$), tenemos que:

$$s[0] = j \neq (j+1) \pmod 2 = s[n-1]$$

- Si $0 \neq h < k$, tenemos que:

$$s[h] = j = j = s[h]$$

- Si $0 \neq h > k$, tenemos:

$$s[h] = (j + 1) \bmod 2 = (j + 1) \bmod 2 = s[h]$$

En cualquier caso, P_h no puede entrar en la sección crítica, con lo que P_k es el único proceso que puede entrar en sección crítica.

- Supuesto que no hay ningún proceso distinguido, llegamos a contradicción con la Proposición anterior, ya que inicialmente P_0 es distinguido y tras cada ejecución de la sección crítica tenemos que esta propiedad cicla entre los procesos, con lo que siempre habrá un proceso distinguido.
- (ii) Gracias a la anterior Proposición, sabemos que si nos fijamos en un proceso P_i en el momento en el que el proceso P_k es distinguido:

- Si $i < k$, dentro de $n - k + i$ turnos, el proceso P_i será distinguido.
- Si $i > k$, dentro de $k - i$ turnos, el proceso P_i será distinguido.
- Si $k = i$, tras la ejecución de la sección crítica por parte de P_i , pasarán n turnos para volver a ser un proceso distinguido.

Por tanto, si todos los n procesos intentan a la vez acceder a la sección crítica, la espera por parte de cada proceso para volver a entrar en ella será de como máximo n turnos.

Ejercicio 5.2.6. Se tienen 2 procesos concurrentes que representan 2 máquinas expendedoras de tickets (en la Figura 5.23) (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.

```

1  {variables compartidas y valores iniciales}
   var n1 : integer := 0;
     n2 : integer := 0;

1  Process P1;
   begin
   while true do begin
     n1 := 1; {E1.1}
5    n1 := n2 + 1; {L1.1; E2.1}
     while n2 <> 0 and {L2.1}
       n2 < n1 do begin end;
       {L3.1}

     {Sección crítica, SC.1}
10    n1 := 0; {E3.1}
     {Resto de senetencias, RS.1}
   end
   end

1  Process P2;
   begin
   while true do begin
     n2 := 1; {E1.2}
5    n2 := n1 + 1; {L1.2; E2.2}
     while n1 <> 0 and {L2.2}
       n1 <= n2 do begin end;
       {L3.2}

     {Sección crítica, SC.2}
10    n2 := 0; {E3.2}
     {Resto de senetencias, RS.2}
   end
   end

```

Figura 5.11: Código para el Ejercicio 5.2.6.

(a) Demostraremos cada uno de las propiedades requeridas:

Exclusión mutua.

Supongamos que el proceso P1 ha conseguido entrar en su sección crítica, y veamos que mientras esto sucede, el proceso P2 no puede entrar también a la sección crítica. Para ello, supongamos pues que P1 ha entrado en la sección crítica. Entonces, tiene que haber sucedido anteriormente un momento en el que no se daba la condición de espera para P1: $n2 \neq 0$ and $n2 < n1$, con lo que habremos tenido alguno de los dos siguientes momentos:

- $n2 = 0$, con lo que cuando P1 entró a la sección crítica, P2 había salido de la suya anteriormente. En este punto, sabemos que $n1 \neq 0$, ya que P1 está dentro de su sección crítica, por lo que tenemos que ver que $n1 \leq n2$, para concluir que P2 no puede entrar a la sección crítica.

Hemos razonado anteriormente que P2 salió de su sección crítica y que luego fue cuando P1 entró a la suya. Si P2 quiere volver a entrar a la sección crítica, debe volver a pasar el protocolo de entrada a la misma, con lo que en algún momento ejecutará la instrucción $n2 := n1 + 1$; y como el valor de la variable $n1$ no es modificada por P1 mientras que este está en la sección crítica, tenemos que $n1 < n1 + 1 = n2$, con lo que P2 no podrá entrar a la sección crítica.

- $n2 \geq n1$, hasta este momento se ha tenido que ejecutar la instrucción $n1 := n2 + 1$, con lo que $n1 > n2$. Sin embargo, como estamos bajo las hipótesis de $n2 \geq n1$, también se habrá tenido que ejecutar

la instrucción $n2 := n1 + 1$. En este momento, tendremos que tanto $n1$ como $n2$ valen 2, con lo que P1 entrará a la sección crítica y P2 no podrá hacerlo, por ser $n1 = n2$.

Supongamos ahora que P2 va a entrar a la sección crítica y vemos si P1 puede entrar tras él. Si P2 ha entrado en la sección crítica es porque en algún momento no se ha cumplido $n1 \neq 0$ and $n1 \leq n2$, por lo que han tenido que darse:

- $n1 = 0$, por un razonamiento análogo al anterior en el caso $n2 = 0$ llegamos a la conclusión de que P1 no puede entrar a la sección crítica.
- $n1 > n2$, en este caso, la instrucción $n2 := n1 + 1$; ya se ha tenido que haber ejecutado, pero en dicho caso tendríamos $n2 > n1$, con lo que sabemos que la instrucción $n1 := n2 + 1$; se tuvo que ejecutar después. Para que P1 entre a la sección crítica con P2, tendría que suceder que $n2 \geq n1$ (es fácil ver que $n2 < 0$), pero esto no puede suceder, porque P1 ya no puede modificar ya ninguna variable. Concluimos que P1 no puede entrar a la sección crítica.

Alcanzabilidad.

- Si el proceso P1 ejecuta el protocolo de entrada mientras que P2 está ocioso ($n2 = 0$), entonces P1 conseguirá alcanzar la sección crítica. Algo similar sucede cuando P1 está ocioso y es P2 quien quiere entrar a la sección crítica.
- Supongamos pues que ambos procesos quieren entrar a la sección crítica. Si ninguno consigue llegar a ella es porque se dan las dos condiciones de espera al mismo tiempo, con lo que tenemos:

$$n2 \neq 0 \wedge n2 < n1 \wedge n1 \neq 0 \wedge n1 \leq n2$$

Es decir, $n1 > n2$ y $n1 \leq n2$ a la vez, lo cual es imposible, con lo que algún proceso conseguirá entrar a la sección crítica.

Ausencia de inanición.

Supongamos que el proceso P1 sufre de inanición porque es continuamente adelantado por el proceso P2. En dicho caso, P1 se mantendrá esperando en su bucle de espera activa bajo la condición $n2 \neq 0$ and $n2 < n1$. Sin embargo, cuando P2 salga de su sección crítica, ejecutará $n2 = 0$, con lo que esta situación (que P2 adelante continuamente a P1) es imposible. Un razonamiento análogo justifica la ausencia de inanición para P2.

- (b) Viendo que en las demostraciones de ausencia de inanición y alcanzabilidad no hemos usado dichas instrucciones, sospechamos que las instrucciones son necesarias para garantizar la exclusión mutua. Pensando un poco, llegamos a la traza de ejecución de la Tabla 5.3, que nos permite meter a P1 y a P2 a la vez en la sección crítica (suponemos que estamos trabajando con el código de la Figura 5.23 donde hemos comentado las instrucciones de las líneas 4).

Línea de P1	Línea de P2	n1	n2
3	3	0	0
3	5	0	1
3	6	0	1
3	9	0	1
5	9	1	1
6	9	1	1
9	9	1	1

Tabla 5.3: Traza de ejecución que lleva a tener dos procesos en SC.

Ejercicio 5.2.7. El siguiente programa (en la Figura 5.24) 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.

```

1  {variables compartidas y valores iniciales}
   var c0 : integer := 1;
       c1 : integer := 1;

```

```

1  Process P0;
   begin
   while true do begin
       repeat
5      c0 := 1 - c1;
       until c1 <> 0;

       {Sección crítica}
       c0 := 1;
10     {Resto de sentencias}
   end
   end

```

```

1  Process P1;
   begin
   while true do begin
       repeat
5      c1 := 1 - c0;
       until c0 <> 0;

       {Sección crítica}
       c1 := 1;
10     {Resto de sentencias}
   end
   end

```

Figura 5.12: Código para el Ejercicio 5.2.7.

El algoritmo de la Figura 5.24 no es una solución correcta al problema de la exclusión mutua, debido a que la instrucción $x := 1 - y$ no es atómica, con lo que se podría suceder la traza de ejecución de P0 y P1 de la Tabla 5.4 (donde indicamos en cada caso la línea de la instrucción que se ejecuta, siendo 1(5) la lectura de $c1$ en P0 y e(5) la escritura en $c0$ de P0, de forma análoga con P1), que nos lleva a tener tanto a P0 como a P1 en la sección crítica a la vez, con lo que no cumple con la propiedad de seguridad exigida.

Línea de P0	Línea de P1	c0	c1
4	4	1	1
1(5)	4	1	1
1(5)	1(5)	1	1
e(5)	1(5)	0	1
e(5)	e(5)	0	0
6	6	0	0
8	8	0	0

Tabla 5.4: Traza de ejecución que lleva a tener dos procesos en SC.

Ejercicio 5.2.8. 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.

No sería posible, ya que contradiría el Lema 3 que demostramos en para demostrar las propiedades del algoritmo de Peterson para N procesos:

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

Ejercicio 5.2.9. 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.

Falso, en la teoría vimos que el número máximo de turnos que tiene que esperar cualquier proceso para entrar en sección crítica es de $\frac{N(N-1)}{2}$ turnos.

Ejercicio 5.2.10. Con respecto al algoritmo de la Figura 5.27 (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.

Para demostrar la falsedad de la afirmación basta con dar una traza de ejecución en la que esta no se cumpla. Para ello, supongamos que tenemos dos procesos, P0 y P1 ejecutando el protocolo de entrada de la sección crítica, de forma que se da la siguiente traza de ejecución:

- Ambos ejecutan la instrucción de la línea 9, cambiando su flag a **solicitando**.
- Ambos consultan que su identificador (**i**) es distinto del turno, con lo que ejecutan el bucle, viendo ambos que **flag[turno] = pasivo**.
- Ambos modifican el valor de la variable compartida **turno**, de forma que es el proceso P1 quien modifica dicha variable en último lugar.

```
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         while turno <> i do begin
                if flag[turno] = pasivo then
                    turno := i;
                end
            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 5.13: Código para el Ejercicio 5.2.10.

- Mientras P1 terminaba de modificar la variable **turno**, a P0 le da tiempo a cambiar su flag a **enSC** y de ejecutar el bucle de la línea 19, con lo que le da tiempo a comprobar que ningún otro proceso tiene su flag a **enSC**, saliendo del bucle con un valor de $j = N$, lo que le permite entrar en la sección crítica.
- P0 y P1 intentaron pasar simultáneamente el primer bucle, con el proceso del turno en estado pasivo, P1 fue el último en asignar la variable **turno** y al final P0 fue el primer proceso que consiguió entrar en la sección crítica.

Ejercicio 5.2.11. El algoritmo de la Figura 5.14 (algoritmo de Knuth para N procesos) resuelve el problema de la exclusión mutua para N procesos, para 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 $p(0)$.
- (a) Todas estas propiedades fueron ya demostradas en los apuntes de teoría.
 - (b) Para ello, describiremos a continuación un escenario en el que los procesos P1 y P2 consiguen pasar el bucle de la línea 13 (es decir, llegar a la línea 21), suponiendo que **turno** = 0. El escenario es el contemplado en la Tabla 5.5, donde suponemos que en todo momento el proceso P0 está pasivo, es decir, **flag**[0] = **pasivo**.

Línea de P1	Línea de P2	j_1	j_2
12	12	0	0
13	13	0	0
14	14	0	0
17	14	2	0
13	14	2	0
14	14	2	0
17	14	1	0
13	17	1	2
21	13	1	2
21	21	1	2

Tabla 5.5: Traza de ejecución en la que 2 procesos pasan el bucle de la línea 13.


```
1  var flag : array[0..N-1] of (pasivo, solicitando, enSC);
    turn := 0..N-1;
    flag := pasivo;
    turn := 0;
5
    Process P(i);
    var j : integer;
    begin
        {Resto de instrucciones}
10    repeat
        flag[i] := solicitando;
        j := turn;
        while j <> i do begin
            if flag[j] <> pasivo then
15                j := turn;
            else
                j := (j - 1) mod N;
            endif;
        end
20
        flag[i] := enSC;
        j := 0;

        while (j < N) and ((j=i) or flag[j] <> enSC) do begin
25            j := j + 1;
        end
    until (j >= N);

    turn := i;
30    {Seccion crítica}
    j := (turn + 1) mod N;
    turn := j;
    flag[i] := pasivo;
end
```

Figura 5.14: Código para el Ejercicio 5.2.11

Ejercicio 5.2.12. Si en el algoritmo de Dijkstra de la Figura 5.27 se cambia la instrucción de la línea 12 por esta otra: `if (flag[turno] <> enSC)`, entonces el algoritmo dejaría de ser correcto. Indicar qué propiedad(es) de corrección faltaría(n) y justificar por qué.

El algoritmo de Dijkstra tiene las propiedades de garantizar la exclusión mutua y de alcanzabilidad. Si observamos la demostración de la propiedad de exclusión mutua que hicimos en teoría, esta no dependía del bucle de la línea 11, con lo que su modificación no dará lugar a que más de un proceso ejecute la sección crítica al mismo tiempo.

Por tanto, podemos pensar que la única propiedad de corrección que faltaría sería la de alcanzabilidad de la sección crítica. Para ver que esta no se da, supongamos un escenario en el que tenemos dos procesos, P0 y P1 ejecutando el protocolo de entrada a la sección crítica del código modificado, de forma que se da la traza de ejecución de la Tabla 5.6. Si la traza del programa continúa siendo las que aparecen entre las dos instrucciones destacadas, estas se repetirían de forma infinita, dándose un interbloqueo entre los dos procesos, ya que después de que uno cambie el turno, el otro proceso comprueba que no tiene el turno y lo cambia antes de que el que lo cambió vea que el turno es igual a su identificador.

Al darse una situación de interbloqueo, el algoritmo no cuenta con la propiedad de alcanzabilidad.

Línea de P0	Línea de P1	turno
9	9	2
11	11	2
12	11	2
13	11	0
13	12	0
13	13	1
11	13	1
12	13	1
13	13	0
13	11	0

Tabla 5.6: Traza de ejecución que da lugar a un interbloqueo.

Ejercicio 5.2.13. Si en el algoritmo de Knuth de la Figura 5.14 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 <> turno) 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 5.2.14. Demostrar que las instrucciones entre las líneas 18 y 28 del algoritmo de exclusión mutua distribuido de Ricart-Agrawala (de la Figura 5.15) no necesitan ser protegidas dentro de la sección crítica definida por las operaciones `wait()`, `signal()` del semáforo `s`.

```

1  var token_presente : boolean := false;
    enSC : boolean := false;
    peticion : array[1..n] of boolean := false;

1  Process P(i);
  begin
    wait(s);
    if not token_presente then begin
5     broadcast(pet, i);
      receive(acceso);
      token_presente := true;
    end

10   enSC := true;
      signal(s);

      {Sección crítica}

15   enSC := false;
      wait(s);

      for j := i+1 to n, 1 to i-1 do
        if peticion[j] and
20         token_presente then begin

          token_presente := false;
          send(j, acceso);
          peticion[j] := false;
25       end
      end

      signal(s);
  end

1  Process Pet(i);
  begin
    receive(pet, j);
    wait(s);
5    peticion[j] := true;
    if token_presente and not enSC
    then
      {Repetir líneas 18-28}
    end
  end

```

Figura 5.15: Código para el Ejercicio 5.2.14

Ejercicio 5.2.15. Suponer que el algoritmo de Suzuki-Kasami para resolver el problema de la exclusión mutua distribuida para n -procesos se modifica como aparece en la siguiente figura. Explicar por qué dejaría de ser correcto el algoritmo, relacionándolo con cada una de las propiedades de corrección que se demuestran para el algoritmo original.

```

1  var token_presente : boolean := false;
    enSC : boolean := false;
    peticion : array[1..n] of boolean := false;
    {En el algoritmo original -> peticion : array[1..n] of 0..+INF}
5  {además se declara otro array -> token : array[1..n] of 0..+INF}

1  Process P(i);
begin
    wait(s);
    if not token_presente then begin
5      broadcast(pet, i);
        receive(acceso);
        token_presente := true;
    end

10     enSC := true;
        signal(s);

        { SC }

15     enSC := false;
        wait(s);

        for j := i+1 to n, 1 to i-1 do
            if peticion[j] and
20                token_presente then begin
                    token_presente := false;
                    send(j, acceso);
                    peticion[j] := false;
                end
25     end
        signal(s);
end

```

```

1  Process Pet(i);
begin
    receive(pet, j);
    wait(s);
5    peticion[j] := true;
    if token_presente and
        not enSC then begin
        for j := i+1 to n, 1 to i-1 do
            if peticion[j] and
10                token_presente then
                begin
                    token_presente := false;
                    send(j, acceso);
                    peticion[j] := false;
                end
15            end
        end
        signal(s);
    end
end

```

Figura 5.16: Código para el Ejercicio 5.2.15

5.2.2. Monitores

Ejercicio 5.2.16. Sean los procesos P1, P2, y P3, cuyas secuencias de instrucciones son las que se muestran en el cuadro siguiente:

<pre> 1 {variables globales} Process P1; begin while true do 5 begin a; b; c; end 10 end </pre>	<pre> 1 Process P2; begin while true do 5 begin d; e; f; end 10 end </pre>	<pre> 1 Process P3; begin while true do 5 begin g; h; i; end 10 end </pre>
--	--	--

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:

```

1  a;
   sem_signal(s);
   b;
   c;

```

P2:

```

1  d;
   sem_wait(s);
   e;
   f;

```

P3:

```

1  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:

```

1  a;
   sem_signal(s1);
   b;
   c;

```

P2:

```

1  d;
   sem_wait(s1);
   sem_wait(s2);
   e;
5  f;

```

P3:

```

1  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:

```
1  a;
   b;
   sem_signal(s1);
   sem_signal(s2);
5  c;
```

P2:

```
1  d;
   sem_wait(s1);
   e;
   f;
```

P3:

```
1  g;
   sem_wait(s2);
   h;
   i;
```

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:

```
1  a;
   sem_wait(s);
   b;
   sem_signal(s);
5  c;
```

P2:

```
1  d;
   e;
   sem_wait(s);
   f;
5  sem_signal(s);
```

P3:

```
1  g;
   sem_wait(s);
   h;
   sem_signal(s);
5  i;
```

Ejercicio 5.2.17. 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.

```
1  {variables globales}
   Process P1;
   var m: integer;
   begin
5   while true do
       begin
           m:= 2*x - n;
           print(m);
       end
10  end
```

```
1  Process P2;
   var d: integer;
   begin
5   while true do
       begin
           d:= leer_teclado();
           x:= d - c*5;
       end
10  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:

```
1 while true do begin
    sem_wait(sem_cons);
    m:= 2*x - n;
    sem_signal(sem_prod);
5  print(m);
end
```

P2:

```
1 while true do begin
    d:= leer_teclado();
    sem_wait(sem_prod);
    x:= d - c*5;
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:

```
1 while true do begin
    sem_wait(sem_cons);
    m:= 2*x - n;
    sem_signal(sem_prod);
5  print(m);
    sem_wait(sem_cons);
    sem_signal(sem_prod);
end
```

P2:

```
1 while true do begin
    d:= leer_teclado();
    sem_wait(sem_prod);
    x:= d - c*5;
5  sem_signal(sem_cons);
end
```

De esta forma, el consumidor desperdicia los valores producidos en posición par por el productor.

Ejercicio 5.2.18. 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:

```
1 // Inicialización
int sem_init(sem_t* semaforo, int pcompartido, unsigned int contador);

// Destrucción
5 int sem_destroy(sem_t* semaforo);

// Sincronización-espera
int sem_wait(sem_t* semaforo);

10 // 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 chicos y chicas 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)

5      if(chica_dentro){ // Baño ocupado
           chicos_esperando++;
           sem_post(mutex); // Libera em
           sem_wait(cola_chicos);
           chicos_esperando--;
```

```

10     }

    chicos_dentro++;
    // Si puede entrar otro
    if(chicos_dentro < MAX && chicos_esperando > 0){
15         sem_post(coola_chicos);
    }else{
        sem_post(mutex);
    }
}

20 void sale_chico(){
    sem_wait(mutex);    // Adquiere em
    chicos_dentro--;

25    // Si se queda el baño libre, hay chica esperando pero no chicos
    if(chicos_esperando == 0 && chicas_esperando > 0 && chicos_dentro ==
    0){
        sem_post(coola_chicas);
    }else{
        sem_post(mutex);
30    }
    sem_post(max);
}

```

Ahora, las funciones de entrada y salida de las chicas serían:

```

1 void entra_chica(){
    sem_wait(mutex);    // Adquiere em

    // Si el baño está ocupado o hay un chico esperando
5    if(chicos_dentro > 0 || chica_dentro || chicos_esperando > 0){
        chicas_esperando++;
        sem_post(mutex);
        sem_wait(coola_chicas);
        chicas_esperando--;
10    }

    chica_dentro = true;
    sem_post(mutex);
}

15 void sale_chica(){
    sem_wait(mutex);    // Adquiere em
    chica_dentro = false;

20    if(chicos_esperando > 0){    // Prioridad a chicos
        sem_post(coola_chicos);
    }else if(chicas_esperando > 0){
        sem_post(coola_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);
    }else{
30    }
        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

5     if(chicos_dentro > 0 || chica_dentro){    // Si baño ocupado
        chicas_esperando++;
        sem_post(mutex);
        sem_wait(cola_chicas);

10        chicas_esperando--;
    }

    chica_dentro = true;

```

```

    sem_post(mutex);
}
15 void sale_chica(){
    sem_wait(mutex);    // Adquiere em
    chica_dentro = false;

20    if(chicas_esperando > 0){    // Desbloquea a chica
        sem_post(cola_chicas);
    }else if(chicos_esperando > 0){    // Desbloquea a chico
        sem_post(cola_chicos);
    }else{
25        sem_signal(mutex);
    }
}

```

Ejercicio 5.2.19. 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 5.2.20. 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:

```

1  monitor Recursos (N1, N2 : integer);
    var libres : array[1..2] of integer;
        colas : array[1..2] of condition;

5      begin
        libres[1] = N1;
        libres[2] = N2;
      end

10     procedure pedir_recurso(tipo : 1..2);
      begin
        if libres[tipo] = 0 then
          colas[tipo].wait();
        end;
15        libres[tipo]--;
      end

      procedure liberar_recurso(tipo : 1..2);
      begin
20        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:

```

1  var monitor : Recursos(1,1);

  process P1;
  begin
5    monitor.pedir_recurso(1);
    monitor.pedir_recurso(2);
    {Uso de los recursos}
    monitor.liberar_recurso(1);
    monitor.liberar_recurso(2);
10   end

  process P2;
  begin
15   monitor.pedir_recurso(2);
    monitor.pedir_recurso(1);
    {Uso de los recursos}
    monitor.liberar_recurso(2);
    monitor.liberar_recurso(1);

```

```

end
20 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:

```

1    monitor Recursos (N1, N2 : integer);
    var libres : array[1..2] of integer;
        colas : array[0..2] of condition;

5    begin
        libres[1] := N1;
        libres[2] := N2;
    end

10   procedure pedir_recurso(tipo : 0..2);
    begin
        if tipo = 0 then
            if(libres[1] = 0 or libres[2] = 0) then
                colas[0].wait();

15         end
            libres[1]--;
            libres[2]--;
        else begin
            if(libres[tipo] = 0) then

20         colas[tipo].wait();

            end
            libres[tipo]--;
        end
    end
end

```

```

25  procedure liberar_recurso(tipo : 1..2);
    var otro_tipo = tipo mod 2 + 1;
    begin
        libres[tipo]++;
30      {Si hay otro tipo}
        if(libres[otro_tipo] > 0 and cola[0].queue()) then
            cola[0].signal();
        else
35          cola[tipo].signal();
        end
    end
end
end

```

Ejercicio 5.2.21. 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:

```

1  Monitor LecEsc;
    var lec_dentro : integer;
        esc_dentro : boolean;
        cola_lec, cola_esc : condition;
5
    begin
        lec_dentro = 0;
        esc_dentro = false;
    end
10
    procedure entra_lector();
    begin
        if esc_dentro then

```

```

        cola_lec.wait();
15    end

        lec_dentro++;

        if cola_lec.queue() then {por eficiencia}
20        cola_lec.signal();
        end
    end

    procedure sale_lector();
25    begin
        lec_dentro--;

        if lec_dentro = 0 then
            cola_esc.signal();
30        end
    end

    procedure entra_escritor();
    begin
35    if lec_dentro > 0 OR esc_dentro then
        cola_esc.wait();
        end

        esc_dentro = true;
40    end

    procedure sale_escritor();
    begin
        esc_dentro = false;
45

        if cola_lec.queue() then
            cola_lec.signal();
        else
            cola_esc.signal();
50        end
    end
end

```

2. Solución para dar prioridad a los escritores. En este caso, el código es idéntico, salvo en el procedimiento `sale_escritor`, que quedaría de la siguiente forma:

```

1  procedure sale_escritor();
   begin
       esc_dentro = false;

5   if cola_esc.queue() then
       cola_esc.signal();
   else
       cola_lec.signal();
   end
10 end

```


3. Para esta solución, como queremos mantener el orden FIFO, tenemos que bloquear a todos los procesos en una misma cola. Sin embargo, como al desbloquear un proceso de dicha cola no sabremos si estaremos desbloqueando a un lector o a un escritor, debemos introducir una nueva variable de tipo condición, que contendrá al primer proceso de la cola anterior.

La condición de espera para la cola original es que esta nueva cola no esté vacía. De esta forma:

```

1  Monitor LecEsc;
    var lec_dentro : integer;
        esc_dentro : boolean;
        cola, primero : condition;

5

    begin
        lec_dentro := 0;
        esc_dentro := false;
    end

10

    procedure entra_lector();
    begin
        {si no es el primero, hay que respetar FIFO}
        if primero.queue() then
15            cola.wait();

            if esc_dentro then
                primero.wait();

20            lec_dentro := lec_dentro + 1;
            cola.signal();
        end

    procedure sale_lector();
25    begin
        lec_dentro := lec_dentro - 1;

        if lec_dentro = 0 then
            primero.signal();
30    end

    procedure entra_escritor();
    begin
        if primero.queue() then
35            cola.wait();

            if esc_dentro or lec_dentro > 0 then
                primero.wait();

40            esc_dentro := true;
            cola.signal();
        end

    procedure sale_escritor();
45    begin
        esc_dentro := false;
        primero.signal();

```

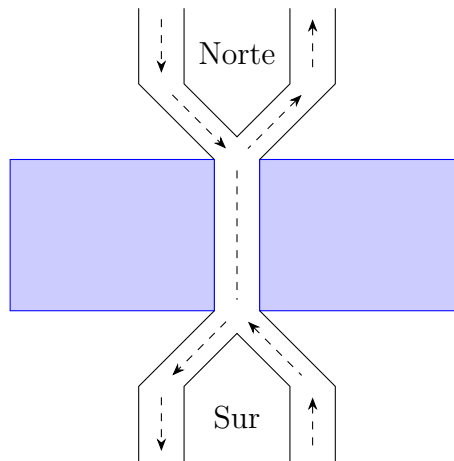


Figura 5.17: Problema de exclusión mutua en el acceso de coches desde 2 sentidos opuestos a un puente de un solo carril.

```
end
end
```

Ejercicio 5.2.22. Varios coches que vienen del norte y del sur pretenden cruzar un puente sobre un río (ver Figura 5.17). 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
10      cola_N.wait();

       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
25      coches_S.signal();
   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
20      coches_N.signal();
   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:

```

1  Monitor Puente;
   var coches_N, coches_S, N_pueden, S_pueden : integer;
       cola_N, cola_S : condition;

5  procedure EntrarCocheDelNorte();
   begin
       { Hay coches en otro sentido }
       if (coches_S > 0 or N_pueden = 0) then cola_N.wait();

10      coches_N := coches_N + 1;

       { Si hay esperando en el sur }
       if cola_S.queue() then N_pueden := N_pueden - 1;

15      if (coches_N.queue() and N_pueden > 0) then coches_N.signal();
   end

   procedure SalirCocheDelNorte();
   begin
20      coches_N := coches_N - 1;

       { El puente se queda vacío }
       if coches_N = 0 then
           begin
25              S_pueden := UMBRAL;
              coches_S.signal();
           end
       end

   procedure EntrarCocheDelSur();
   begin
30       { Hay coches en otro sentido }
       if (coches_N > 0 or S_pueden = 0) then cola_S.wait();

35      coches_S := coches_S + 1;

       { Si hay esperando en el norte }
       if cola_N.queue() then S_pueden := S_pueden - 1;

40      if (coches_S.queue() and S_pueden > 0) then coches_S.signal();
   end

   procedure SalirCocheDelSur();
   begin
45      coches_S := coches_S - 1;

       { El puente se queda vacío }
       if coches_S = 0 then
           begin
50              N_pueden := UMBRAL;
              coches_N.signal();
           end
       end

   begin
55      coches_N := 0; coches_S := 0; N_pueden := UMBRAL; S_pueden := UMBRAL;
   end
end

```

Ejercicio 5.2.23. 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:

```

1  monitor Olla ;
   ....
begin
   ....
5  end;

proceso ProcSalvaje[ i:1..N ] ;
begin
while true do begin
10   Olla.Servirse_1_misionero();
      Comer(); { es un retraso aleatorio }
      end
end;

15 proceso ProcCocinero ;
begin
while true do begin
      Olla.Dormir();
      Olla.Rellenar_Olla();
20   end
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):

```

1  Monitor Olla;
   var num_misioneros : integer;
      comer, dormir : condition;

5  procedure Servirse_1_misionero();
begin
   { Si no quedan misioneros }
   if num_misioneros = 0 then
begin
10   dormir.signal(); { Despierta al cocinero }
      comer.wait();
      end

      num_misioneros := num_misioneros - 1;
15

```

```

    if num_misioneros > 0 then
        comer.signal();
    end

20  procedure Dormir();
    begin
        if num_misioneros > 0 then
            dormir.wait();
        end

25  procedure Rellenar_Olla();
    begin
        num_misioneros := M;

30      { Avisa a un salvaje }
        comer.signal();
    end

    begin
35      num_misioneros := M;
    end
end

```

Ejercicio 5.2.24. 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 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:

```
1  Monitor Cuenta;  
   var saldo : integer;  
       cola : condition;  
  
5  procedure depositar(c : integer);  
   begin  
       saldo = saldo + c;  
       cola.signal();  
   end  
  
10 procedure retirar(c : integer);  
   begin  
       while c > saldo do begin  
           cola.signal();  
15          cola.wait();  
       end do  
       saldo = saldo - c;  
       cola.signal();  
   end  
  
20 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**:

```
1  procedure retirar(c : integer);  
   begin  
       while c > saldo do begin  
           cola.wait(c);  
5          end do  
       saldo = saldo - c;  
       cola.signal();  
   end
```

2. Ahora, con orden en las retiradas, es necesario disponer de una variable condición más, que controle si hay alguien esperando a retirar dinero.

- a) Con colas normales FIFO en las variables condición, es necesario introducir una nueva variable condición, **ventanilla**, que almacene el primer proceso que quiere retirar saldo, con la finalidad de que no se adelanten entre sí.

```
1  Monitor Cuenta;
   var saldo : integer;
       cola, ventanilla : condition;

5  procedure depositar(c : integer);
   begin
       saldo = saldo + c;
       ventanilla.signal();
   end

10 procedure retirar(c : integer);
   begin
       if ventanilla.queue() then
           cola.wait();

15       while c > saldo do begin
           ventanilla.wait();
       end do

20       saldo = saldo - c;
       cola.signal();
   end

   begin
25       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.

```
1  Monitor Cuenta;
   var saldo : integer;
       contador : integer;
       cola : condition;

5  procedure depositar(c : integer);
   begin
       saldo = saldo + c;
       cola.signal();

10  end

   procedure retirar(c : integer);
   var ticket : integer;
   begin

15       ticket = contador;
       contador = contador + 1;

       if cola.queue() then
           cola.wait(ticket);

20       while c > saldo do
           cola.wait(ticket);
       end
```



```

25     saldo = saldo - c;
        cola.signal();
    end

    begin
30     saldo = 0;
        contador = 0;
    end
end

```

Ejercicio 5.2.25. Los procesos P_1, P_2, \dots, P_n comparten un único recurso R , pero sólo un proceso puede utilizarlo cada vez. Un proceso P_i 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 P_i tiene prioridad i , (con $1 \leq i \leq n$), donde los números menores implican mayor prioridad (es decir, si $i < j$, entonces P_i pasa por delante de P_j). 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 hay variables condición con colas prioritarias.

El código sería el siguiente:

```

1  Monitor Recurso;
    var libre : boolean;
        cola : condition; {prioritaria}

5  begin
    libre := false;
    end

    procedure liberar();
10  begin
    libre := true;
    cola.signal();
    end

15  procedure pedir(id_proceso : 1..n);
    begin
    if not libre then
        cola.wait(id_proceso);

20  libre := false;
    end
end

```

Suponiendo que no hay variables condición con colas prioritarias.

Para ello, dispondremos de un array de booleanos, `peticion`, de forma que si

`peticion[i] = true`, entonces el proceso P_i estará bloqueado solicitando un recurso:

```

1  Monitor Recurso;
    var libre : boolean;
        peticion : array[1..n] of boolean;
        cola : condition;

5

    procedure liberar();
    var j : integer := 0;
    begin
        {Buscamos peticiones por orden de prioridad}
10       repeat
            j := j + 1;
        until(j = n or peticion[j]);

        if peticion[j] then {j es el de mayor prioridad que pide el recurso}
15       begin
            peticion[j] := false; {Le cedemos el recurso directamente}
            cola.signal();
        else
            libre := true;
20       end
    end

    procedure pedir(id_proceso : 1..n);
    begin
25       if not libre then
        begin
            {Bloqueamos al proceso}
            peticion[id_proceso] := true;
            while peticion[id_proceso] do begin
30                 cola.signal();
                 cola.wait();
            end
        end

        libre := false;
35     end

    begin
        libre := false;
40     peticion := (false, false, ..., false);
    end
end

```

Complicando el problema.

Si ahora suponemos que no hay un solo recurso R , sino que tratamos de resolver el problema para m recursos, disponemos del siguiente monitor, que cuenta con dos arrays de booleanos:

- `libre`, de forma que si `libre[i] = true`, entonces el recurso i -ésimo estará libre.
- `peticion`, con el mismo comportamiento que antes.

```

1  Monitor Recurso;
    var libre : array[1..m] of boolean;
      peticion : array[1..n] of boolean;
      I : integer;
5    cola : condition;

    procedure liberar(id_recurso : 1..m);
      var j : integer := 0;
      begin
10      {Buscamos peticiones por orden de prioridad}
        repeat
          j := j + 1;
        until(j = n or peticion[j]);

15      {Si había alguna petición}
        if peticion[j] then
          begin
            {Le cedemos el recurso}
            I := id_recurso;
20          peticion[j] := false;
            cola.signal();
          else {El recurso queda libre}
            libre[id_recurso] := true;
          end
25      end

      {devuelve el recurso asignado}
      procedure pedir(id_proceso : 1..n) : integer;
        var k : integer := 0;
30      begin
        {Buscamos recursos libres}
        repeat
          k := k + 1;
        until(k = m or libre[k]);
35      {Si no había recursos libres}
        if not libre[k] then
          begin
            peticion[id_proceso] := true;
40          while peticion[id_proceso] do begin
              cola.signal();
              cola.wait();
            end
          else
45          I := k;
          end

          libre[I] := false;
          return I;
50      end

      begin
        libre := (true, true, ..., true);
        peticion := (false, false, ..., false);
55      end
    end

```

Ejercicio 5.2.26. 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.

```

1  Monitor Barrera2 ;
   var n : integer; { num. de proc. que han llegado desde el signal }
   s : condicion ; { cola donde espera el segundo }
   procedure entrada() ;
5     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: }
10          n := 0; { inicializa el contador }
           s.signal() { despertar al primero }
       end
       end
15     { Inicializacion }
       begin
           n := 0 ;
       end
   end
end

```

Para realizar esta implementación, haremos uso de las siguientes variables compartidas:

```

1  var n : integer := 0;
    sem : semaphore := 0;
    mutex : semaphore := 1;

```

Usaremos la siguiente función, que hará las veces del procedimiento **entrada** del monitor:

```

1  procedure entrada();
   begin
       sem_wait(mutex); {Adquiere em}
       n := n + 1;
5
       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}
10          n := 0;
           sem_signal(sem);
           sem_signal(mutex);
       end
   end
end

```

Ejercicio 5.2.27. 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 $i + 1 \bmod 5$. El monitor **MonFilo** exportará dos procedimientos: `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:

```

1  monitor MonFilo ;
   ....
   procedure coge_tenedor( num_ten, num_proc : integer );
   ....
5  procedure libera_tenedor( num_ten : integer );
   ....
   begin
   ....
   end
10 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}
15           comer();
           MonFilo.libera_tenedor(i);
           MonFilo.libera_tenedor(i+1 mod 5);
           pensar();
       end
20   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. En primer lugar, vemos que una de las variables permanentes del monitor ha de ser el array `tenedor_ocupado`, donde `tenedor_ocupado[i]=true` indica que el tenedor i -ésimo ha sido cogido. Por otro lado, necesitamos un array de variables condición, `cola_tenedor`, en cuya cola esperen los procesos filósofos que necesitan del tenedor i -ésimo.

```

1  monitor MonFilo;
    var   tenedor_ocupado : array[0..4] of boolean;
        cola_tenedor : array[0..4] of condition;

5  procedure coge_tenedor(num_tenedor, num_proceso : integer);
begin
    { Si ya han cogido el tenedor }
    if tenedor_ocupado[num_tenedor] then
        cola_tenedor[num_tenedor].wait();

10     tenedor_ocupado[num_tenedor] := true;
    end

    procedure libera_tenedor(num_tenedor : integer);
15 begin
        tenedor_ocupado[num_tenedor] := false;
        cola_tenedor[num_tenedor].signal();
    end

20 begin
    for i = 0 to 4 do
        tenedor_ocupado[i] := false;
    end
end
25 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 5.7: 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.

3. Para que el monitor no de lugar a un interbloqueo de los procesos, es necesario introducir un nuevo array, que controle qué filósofos disponen de un único tenedor, `un_tenedor`; así como un entero que controle el número de filósofos que tienen un único tenedor, `fil_1t`; y una variable condición `cola`, que nos permita bloquear a los filósofos que traten de coger el 5º tenedor y bloquear al resto:

```

1  monitor MonFilo;
    var   tenedor_ocupado : array[0..4] of boolean;
        un_tenedor : array[0..4] of boolean;
        fil_1t : integer;

5      cola_tenedor : array[0..4] of condition;
        cola : condition;

    procedure libera_tenedor(num_tenedor : integer);
10  begin
        tenedor_ocupado[num_tenedor] := false;
        cola_tenedor[num_tenedor].signal();
    end

15  procedure coge_tenedor(num_tenedor, num_proceso : integer);
    begin
        {Si estamos a punto de bloquear todos los procesos}
        if not un_tenedor[num_proceso] and fil_1t = 4 then
            cola.wait();

20      {En este punto, no hay riesgo de interbloqueo. O bien porque no hay
        4 filósofos con un solo tenedor, o bien porque el proceso actual ya
        tiene un tenedor}
        if tenedor_ocupado[num_tenedor] then
            cola_tenedor[num_tenedor].wait();

25      tenedores[num_tenedor] := true;

        {Hay que actualizar las nuevas variables}
        if not un_tenedor[num_proceso] then {Si antes no tenía tenedor}
            un_tenedor[num_proceso] := true;
            fil_1t := fil_1t + 1;
30      else then {Si ahora tengo dos tenedores}
            un_tenedor[num_proceso] := false;
            fil_1t := fil_1t - 1;

35      { Ya no hay riesgo de interbloqueo, porque este filósofo
        liberará ambos tenedores }
        if fil_1t < 4 then
            cola.signal();
        end
    end
end

40  begin
    for i = 0 to 4 do
        begin
            tenedor_ocupado[i] := false;
            un_tenedor[i] := false;
45      end
    end

    fil_1t := 0;
50  end
end

```

Ejercicios extras

Ejercicio 5.2.28. En un pueblo existe una pequeña barbería con una puerta de entrada y otra de salida, dichas puertas son tan estrechas que solo permiten el paso de una persona cada vez que son utilizadas. En la barbería hay un número indeterminado de sillas (suponemos que siempre hay sillas libres porque los barberos no hacen esperar demasiado a los clientes) de 2 tipos:

- (a) Sillas donde los clientes esperan a los barberos (sillas de espera).
- (b) Sillas de barbero donde se realiza el corte de pelo a los clientes por parte de los barberos.

No hay espacio suficiente para que más de una persona (barbero o cliente) pueda moverse dentro de la barbería en cada momento, por ejemplo, si los clientes se dan cuenta de que ha entrado un barbero, entonces solo un cliente puede levantarse y dirigirse a una silla de tipo (b); un barbero, por su parte, no podría moverse para ir a cortar el pelo hasta que el cliente se hubiera sentado.

Cuando entra un cliente en la barbería, puede hacer una de estas 2 cosas:

- Aguarda en una silla de tipo (a) y espera a que haya barberos disponibles para poder ser atendido, cuando sucede esto último, el cliente se levanta y vuelve a sentarse, esta vez en una silla de tipo (b), para esperar hasta que termine su corte de pelo.
- Se sienta directamente en una silla de tipo (b), si hay barberos disponibles. Un cliente no se levanta de la silla del barbero hasta que este le avisa abriéndole la puerta de salida.

Cuando entra en la barbería, un barbero aguarda a que haya clientes sentados en una silla de tipo (b) esperando su corte de pelo. Después, revisa el estado de los clientes siguiendo un orden numérico establecido, hasta que encuentra un cliente que espera ser atendido; cuando lo encuentra comienza a cortar el pelo y él mismo pasa a estar ocupado. Cuando termina con un cliente, le abre la puerta de salida y espera a que el cliente le pague, después sale a la calle para refrescarse. El barbero no podrá entrar de nuevo en la barbería hasta que haya cobrado al cliente que justamente acaba de atender. No se admite que un cliente pague a un barbero distinto del que cortó el pelo.

```

1 monitor Peluqueria;
   var pelando : array[0..+INF] of (silla_vacia, silla_pelando,
   silla_esperando);
   pagado : array[0..+INF] of boolean;
   clientes_esperando, clientes_pelando, barberos_ociosos, puerta_salida :
   condition;

5
   begin
   pelando := (silla_vacia, silla_vacia, ...);
   pagado := (false, false, ...);
10  end

```



```

{Procedimiento para pelar a los clientes}
procedure pelar(i : integer);
begin
  {Esperar hasta que haya un barbero disponible}
15  if not barberos_ociosos.queue() then
    clientes_esperando.wait();

  {Hay un barbero disponible}
  pelando[i] := silla_esperando; {Se sienta en una silla}
20  barberos_ociosos.signal();

  {Espera su corte de pelo (hasta que el barbero le levante)}
  while pelando[i] <> silla_vacia do begin
    clientes_pelando.signal();
25    clientes_pelando.wait();
  end

  {Espera a ser cobrado}
  pagado[i] := false;
30  while not pagado[i] do begin
    puerta_salida.signal();
    puerta_salida.wait();
  end
end

35 {Procedimiento para barbero para pasar al siguiente cliente}
procedure siguiente_cliente() : integer;
var k := 0;
begin
40  {Si hay clientes esperando, los despertamos}
  if clientes_esperando.queue() then
    clientes_esperando.signal();
  else {Si no hay clientes esperando, esperamos clientes}
    barberos_ociosos.wait();
45  end

  {Hay un cliente esperando a ser pelado en la silla, lo buscamos}
  while pelando[k] <> silla_esperando do begin
    k := k + 1;
50  end
  pelando[k] := silla_pelando;
  return k;
end

55 {Procedimiento para terminar de pelar al cliente}
procedure terminar(i : integer);
begin
  {Termina de pelar al cliente}
  pelando[i] := silla_vacia;
60  clientes_pelando.signal();

  {Cobra al cliente}
  pagado[i] := true;
  puerta_salida.signal();
65  end
end

```

Ejercicio 5.2.29. Suponer un garaje de lavado de coches con dos zonas: una de espera y otra de lavado de coches con 100 plazas de lavado. Un coche entra en la zona de lavado de garaje solo si hay (al menos) 1 plaza libre, si no se queda en la cola de espera. Si un coche entra en la zona de lavado, busca una plaza libre y espera hasta que es atendido por un empleado del garaje. Los coches no pueden volver a entrar al garaje hasta que el empleado que les atendió les cobre el servicio. Suponemos que hay más de 100 empleados que lavan coches, cobran, salen y vuelven a entrar al garaje. Cuando un empleado entra en el garaje comprueba si hay coches esperando ser atendidos, es decir, ya ubicados en su plaza de lavado; si no, aguarda a que haya alguno en esa situación. Si hay al menos un coche esperando, recorre las plazas de la zona de lavado hasta que lo encuentra, entonces los lava, le avisa de que puede salir y, por último, espera a que le pague. Puede suceder que varios empleados hayan terminado de lavar sus coches y estén todos esperando el pago de sus servicios, pero no se admite que un empleado cobre a un coche distinto del que ha lavado. También hay que evitar que al entrar 2 o más empleados a la zona de lavado, habiendo comprobado que hay coches esperando, seleccionen a un mismo coche para lavarlo. Se pide: programar una simulación de la actuación de los coches y de los empleados del garaje, utilizando un monitor con señales urgentes (SU). Se valorará más aquella solución al problema que utilice un número menor de variables condición.

5.3. Paso de mensajes

Ejercicio 5.3.1. En un sistema distribuido, 6 procesos clientes necesitan sincronizarse de forma específica para realizar cierta tarea, de forma que dicha tarea sólo podrá ser realizada cuando tres procesos estén preparados para realizarla. Para ello, envían peticiones a un proceso controlador del recurso y esperan respuesta para poder realizar la tarea específica. El proceso controlador se encarga de asegurar la sincronización adecuada. Para ello, recibe y cuenta las peticiones que le llegan de los procesos, las dos primeras no son respondidas y producen la suspensión del proceso que envía la petición (debido a que se bloquea esperando respuesta) pero la tercera petición produce el desbloqueo de los tres procesos pendientes de respuesta. A continuación, una vez desbloqueados los tres procesos que han pedido (al recibir respuesta), inicializa la cuenta y procede cíclicamente de la misma forma sobre otras peticiones. El código de los procesos clientes aparece aquí abajo. Los clientes usan envío asíncrono seguro para realizar su petición, y esperan con una recepción síncrona antes de realizar la tarea:

```

1 process Cliente[ i : 0..5 ];
  begin
    while true do begin
      send(peticion, Controlador);
5      receive(permiso, Controlador);
      Realiza_tarea_grupal();
    end
  end
end

```

Figura 5.18: Código para el Ejercicio 5.3.1.

Describir en pseudocódigo el comportamiento del proceso controlador, utilizando una orden de espera selectiva que permita implementar la sincronización requerida entre los procesos. Es posible utilizar una sentencia del tipo `select for i=... to ...` para especificar diferentes ramas de una sentencia selectiva que comparten el mismo código dependiente del valor de un índice `i`.

```

1 process Controlador;
  var contador : integer := 0;
    necesarios : integer := 3;
    esperando : array[0..1] of 0..5;
5  begin
    while true do begin
      select
        for i := 0 to 5
          when receive(valor, Cliente[i]) do
10             if contador < necesarios-1 then begin
                esperando[contador] := i;
                contador := contador + 1;
              else
15                 contador := 0;
                send(valor, Cliente[i]);
                send(valor, Cliente[esperando[0]]);
            end
          end
        end
      end
    end
  end
end

```

```

                send(valor, Cliente[esperando[1]]);
            end
        end
    end select
end do
end

```

Ejercicio 5.3.2. En un sistema distribuido, 3 procesos productores producen continuamente valores enteros y los envían a un proceso buffer que los almacena temporalmente en un array local de 4 celdas enteras para ir enviándoselos a un proceso consumidor. A su vez, el proceso buffer realiza lo siguiente, sirviendo de forma equitativa al resto de procesos:

- (a) Envía enteros al proceso consumidor siempre que su array local tenga al menos dos elementos disponibles.
- (b) Acepta envíos de los productores mientras el array no esté lleno, pero no acepta que cualquier productor pueda escribir dos veces consecutivas en el búfer.

El código de los procesos productor y consumidor es el siguiente, asumiendo que se usan operaciones síncronas:

```

1 process Productor [ i : 0..2 ];
  var dato : integer;
  begin
    while true do begin
5      dato := Producir();
      send(dato, Buffer);
    end
  end
end

```

```

1 process Consumidor;
  begin
    while true do begin
      receive(dato, Buffer);
5      Consumir(dato);
    end
  end
end

```

Figura 5.19: Código para el Ejercicio 5.3.2.

Describir en pseudocódigo el comportamiento del proceso **Buffer**, utilizando una orden de espera selectiva que permita implementar la sincronización requerida entre los procesos.

```

1 process Buffer;
  var buffer : array[0..3] of integer;
    primera_libre, primera_ocupada : integer := 0, 0;
    ocupadas : integer := 0;
5    ult_productor : integer := -1;
    dato : integer;
  begin
    while true do begin
      select
10      when ocupadas >= 2 do
        dato := buffer[primera_ocupada];

```

```

    primera_ocupada := (primera_ocupada + 1) mod 4;
    ocupadas := ocupadas - 1;
    send(dato, Consumidor);
15  end
    for i := 0 to 2
        when ocupadas < 4 and i <> ult_productor receive(dato, Productor[i])
            ult_productor := i;
            buffer[primera_libre] := dato;
20      primera_libre := (primera_libre + 1) mod 4;
            ocupadas := ocupadas + 1;
        end
    end
    end
25  end
end

```

Ejercicio 5.3.3. Suponer un proceso productor y 3 procesos consumidores que comparten un buffer acotado de tamaño B. Cada elemento depositado por el proceso productor debe ser retirado por todos los 3 procesos consumidores para ser eliminado del buffer. Cada consumidor retirará los datos del buffer en el mismo orden en el que son depositados, aunque los diferentes consumidores pueden ir retirando los elementos a ritmo diferente unos de otros. Por ejemplo, mientras un consumidor ha retirado los elementos 1, 2 y 3, otro consumidor puede haber retirado solamente el elemento 1. De esta forma, el consumidor más rápido podría retirar hasta B elementos más que el consumidor más lento. Describir en pseudocódigo el comportamiento de un proceso que implemente el buffer de acuerdo con el esquema de interacción descrito usando una construcción de espera selectiva, así como el del proceso productor y de los procesos consumidores. Comenzar identificando qué información es necesario representar, para después resolver las cuestiones de sincronización. Una posible implementación del buffer mantendría, para cada proceso consumidor, el puntero de salida y el número de elementos que quedan en el buffer por consumir:

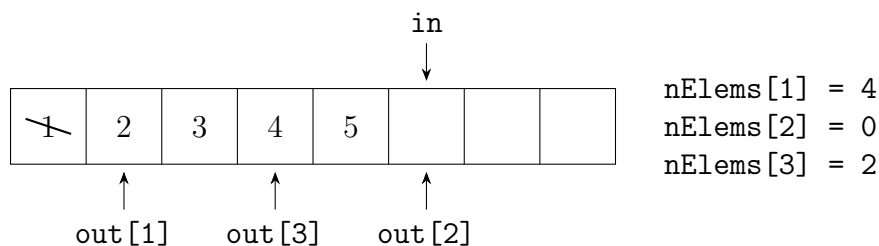


Figura 5.20: Dibujo para el Ejercicio 5.3.3.

En primer lugar, describimos los códigos de los procesos productor y consumidores, por ser estos mucho más fáciles que el del proceso intermedio que usaremos para comunicar ambos tipos de procesos, al no requerir estos de sincronización ninguna. Tanto productor como consumidores mandan mensajes al Buffer, y los consumidores esperan una respuesta del mismo.

```

1 process Productor;
  var dato : integer;
  begin
    dato := Producir();
5    send(dato, Buffer);
  end

```

```

1 process Consumidor[ i : 0..2 ];
  var dato : integer;
  begin
    send(peticion, Buffer);
5    receive(dato, Buffer);
    Consumir(dato);
  end

```

Ahora, desarrollamos el código del proceso Buffer, donde `out[i]` indica la siguiente posición a leer del consumidor *i*-ésimo y `nElems[i]` indica la cantidad de datos que quedan por leer al consumidor *i*-ésimo.

```

1 process Buffer;
  var buffer : array[0..B-1] of integer;
    ocupados : integer := 0;
    in : 0..B-1 := 0;
5  out : array[0..2] of 0..B-1 := (0, 0, 0);
  nElems : array[0..2] of 0..B-1 := (0, 0, 0);
  dato : integer;
  begin
    while true do begin
10      select
        when ocupados < B receive(dato, Productor) do
          buffer[in] := dato;
          in := (in + 1) mod B;
          ocupados := ocupados + 1;
15
          for i:= 0 to 2 do
            nElems[i] := nElems[i] + 1;
          end
          for i := 0 to 2
20            when nElems[i] > 0 receive(peticion, Consumidor[i]) do
              dato := buffer[out[i]];
              out[i] := out[i] + 1 mod B;

              { Índices de los otros consumidores }
25              j := (i+1) mod 3;
              k := (j+1) mod 3;

              { Si nElems es el mayor, era el último en consumir }
              if nElems[i] > nElems[j] and nElems[i] > nElems[k] then
30                ocupados := ocupados - 1;
              end

              nElems[i] := nElems[i] - 1;
              send(dato, Consumidor[i]);
35            end
          end
        end
      end
    end
  end

```

Ejercicio 5.3.4. 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.

```

1 process Salvaje[ i : 0..2 ];
  var peticion : integer := ... ;
  begin
    while true do begin
5      { esperar a servirse un
        misionero }
        ...
        s_send(peticion, Olla);
        { comer }
        Comer();
10    end
  end
end

1 process Cocinero;
  begin
    while true do begin
      { dormir esperando solicitud
        para rellenar }
5      ...
      {confirmar que se ha rellenado
        la olla}
      ...
    end
  end
end

```

Figura 5.21: Código para el Ejercicio 5.3.4.

Implementar los procesos salvajes y cocinero usando paso de mensajes, usando un proceso olla que incluye una construcción de espera selectiva que sirve peticiones de los salvajes y el cocinero para mantener la sincronización requerida, teniendo en cuenta que:

- La solución no debe producir interbloqueo.
- Los salvajes podrán comer siempre que haya comida en la olla.
- Solamente se despertará al cocinero cuando la olla esté vacía.

Mostramos primero los códigos para los procesos salvajes y cocinero:

```

1 process Salvaje[ i : 0..2 ];
  begin
    while true do begin
      s_send(peticion, Olla);
5      Comer();
    end
  end
end

1 process Cocinero;
  begin
    while true do begin
      receive(peticion, Olla);
5      send(rellenar, Olla);
    end
  end
end

```

A continuación, mostramos el código del proceso Olla:

```

1 process Olla;
  var misioneros : integer := 0;
  begin
    while true do begin
5      select
        when misioneros = 0 do

```

```

        send(valor, Cocinero);
        receive(valor, Cocinero);
        misioneros := M;
10      end
      for i := 0 to 2
        when misioneros > 0 receive(valor, Salvaje[i]) do
          misioneros := misioneros - 1;
          end
15    end
  end
end
end

```

Ejercicio 5.3.5. Considerar un conjunto de N procesos, $P[i]$, ($i = 0, \dots, N - 1$) que se pasan mensajes cada uno al siguiente (y el primero al último), en forma de anillo. Cada proceso tiene un valor local almacenado en su variable local `mi_valor`. Deseamos calcular la suma de los valores locales almacenados por los procesos de acuerdo con el algoritmo que se expone a continuación.

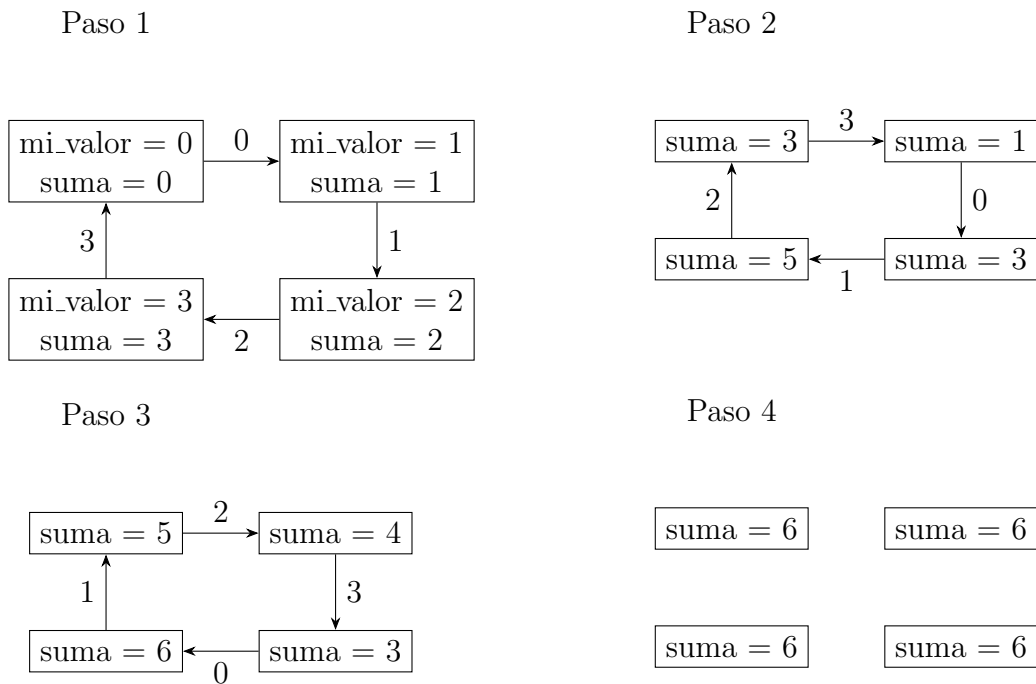


Figura 5.22: Dibujo para el Ejercicio 5.3.5.

Los procesos realizan una serie de iteraciones para hacer circular sus valores locales por el anillo. En la primera iteración, cada proceso envía su valor local al siguiente proceso del anillo, al mismo tiempo que recibe del proceso anterior el valor local de éste. A continuación acumula la suma de su valor local y el recibido desde el proceso anterior. En las siguientes iteraciones, cada proceso envía al siguiente proceso siguiente el valor recibido en la anterior iteración, al mismo tiempo que recibe del proceso anterior un nuevo valor. Después acumula la suma. Tras un total de $N - 1$ iteraciones, cada proceso conocerá la suma de todos los valores locales

de los procesos. Dar una descripción en pseudocódigo de los procesos siguiendo un estilo SPMD y usando operaciones de envío y recepción síncronas:

```

1 process P[ i : 0..N-1 ];
  var mi_valor : integer := ...; {valor aleatorio, igual a i en la figura}
    suma : integer := mi_valor;
  begin
5    for j := 0 to N-1 do begin
      ...
    end
  end
end

```

La solución la podemos encontrar en el siguiente código, donde debemos tener cuidado de que no todos los procesos primero envíen y luego reciban, ya que esto puede dar lugar a una situación de interbloqueo por usar envíos y recepciones síncronas. Para ello, obligamos a que al menos un proceso primero reciba y luego envíe, o viceversa:

```

1 process P[ i : 0..N-1 ];
  var mi_valor : integer := ...; {valor aleatorio, igual a i en la figura}
    suma : integer := mi_valor;
    siguiente : 0..N-1 := (i+1) mod N;
5    anterior : 0..N-1 := (i-1) mod N;
    recibido : integer;
  begin
    for j := 0 to N-1 do begin
      if i == 0 then begin
10        send(mi_valor, siguiente);
        recv(recibido, anterior);
      else
        recv(recibido, anterior);
        send(mi_valor, siguiente);
15      end
      mi_valor := recibido;
      suma += recibido;
    end
  end
end

```

Ejercicio 5.3.6. Considerar un estanco en el que hay tres fumadores y un estancero. Cada fumador continuamente lía un cigarro y se lo fuma. Para liar un cigarro, el fumador necesita tres ingredientes: tabaco, papel y cerillas. Uno de los fumadores tiene solamente papel, otro tiene solamente tabaco, y el otro tiene solamente cerillas. El estancero tiene una cantidad infinita de los tres ingredientes.

- El estancero coloca aleatoriamente dos ingredientes diferentes de los tres que se necesitan para hacer un cigarro, desbloquea al fumador que tiene el tercer ingrediente y después se bloquea. El fumador seleccionado, se puede obtener fácilmente mediante una función `genera_ingredientes` que devuelve el índice (0, 1, ó 2) del fumador escogido.
- El fumador desbloqueado toma los dos ingredientes del mostrador, desbloqueando al estancero, lía un cigarro y fuma durante un tiempo.

- El estancero, una vez desbloqueado, vuelve a poner dos ingredientes aleatorios en el mostrador, y se repite el ciclo.

Describir una solución distribuida que use envío asíncrono seguro y recepción síncrona, para este problema usando un proceso Estancero y tres procesos fumadores Fumador(i) (con $i = 0, 1, 2$).

```

1 process Estancero;
  var ing : array[0..1] of ingredientes;
  begin
    while true do begin
      { Genera dos ingredientes distintos }
      ing := generaIngredientes();

      select
        for i:=0 to 2
10       when genera_ingredientes(ing) do
          send(ing, Fumador[i]);
          receive(confirmacion, Fumador[i]);
        end
      end select
15    end
  end
end

```

```

1 process Fumador[ i : 0..2 ];
  var ingredientes : array[0..1] of ingredientes;
  begin
    while true do begin
      recv(ingredientes, Estancero);
5      send(confirmacion, Estancero);
      fumar(ingredientes);
    end
  end
end

```

Figura 5.23: Código para el Ejercicio 5.3.6.

Ejercicio 5.3.7. En un sistema distribuido, un gran número de procesos clientes usa frecuentemente un determinado recurso y se desea que puedan usarlo simultáneamente el máximo número de procesos. Para ello, los clientes envían peticiones a un proceso controlador para usar el recurso y esperan respuesta para poder usarlo (véase el código de los procesos clientes). Cuando un cliente termina de usar el recurso, envía una solicitud para dejar de usarlo y espera respuesta del Controlador. El proceso controlador se encarga de asegurar la sincronización adecuada imponiendo una única restricción por razones supersticiosas: nunca habrá 13 procesos exactamente usando el recurso al mismo tiempo.

```

1 process Cli[ i : 0..n ];
  var pet_usar : integer := +1;
    pet_liberar : integer := -1;
    permiso : integer := ...;
5 begin
  while true do begin
    send(pet_usar, Controlador);
    receive(permiso, Controlador);
    Usar_recurso();
10    send(pet_liberar, Controlador);
    receive(permiso, Controlador);
  end
end

```

Figura 5.24: Código para el Ejercicio 5.3.7.

Describir en pseudocódigo el comportamiento del proceso controlador, utilizando una orden de espera selectiva que permita implementar la sincronización requerida entre los procesos. Es posible utilizar una sentencia del tipo **select for i=... to ...** para especificar diferentes ramas de una sentencia selectiva que comparten el mismo código dependiente del valor de un índice *i*.

La solución que planteamos es la siguiente, donde tenemos en cuenta las peticiones de obtención y liberación del recurso que llevan al recurso a ser usado por 13 procesos al mismo tiempo. En dicho caso, guardamos la petición hasta que haya cualquiera otra petición (que puede ser del mismo tipo o distinto).

```

1 process Controlador
  var contador : integer := 0;
    pendiente : integer := 0;
    id_pendiente : integer;
5    peticion : integer;
    permiso : integer := 100;
begin
  while true do begin
    select
10    for i:= 0 to n
      when receive(peticion, Cli[i]) do
        { Si no nos sirve el estado al que llega }
        if contador + pendiente + peticion = 13 then begin
          { Sabemos que pendiente = 0 }
15          pendiente := peticion;
          id_pendiente := i;
        else { Si no, se procesa la petición }
          contador := contador + pendiente + peticion;
          send(permiso, Cli[i]);
20
          { Si había una pendiente también se acepta }
          if pendiente <> 0 then begin
            send(permiso, Cli[id_pendiente]);
            pendiente := 0;
25          end
        end
      end
    end
  end
end

```

```

        end
    end select
end
30 end

```

Ejercicio 5.3.8. En un sistema distribuido, tres procesos **Productor** se comunican con un proceso **Impresor** que se encarga de ir imprimiendo en pantalla una cadena con los datos generados por los procesos productores. Cada proceso productor (**Productor**[i], con $i = 0, 1, 2$) genera continuamente el correspondiente entero i , y lo envía al proceso Impresor.

El proceso Impresor se encarga de ir recibiendo los datos generados por los productores y los imprime por pantalla (usando el procedimiento `imprime(entero)`) generando una cadena de dígitos en la salida. No obstante, los procesos se han de sincronizar adecuadamente para que la impresión por pantalla cumpla las siguientes restricciones:

- Los dígitos 0 y 1 deben aceptarse por el impresor de forma alterna. Es decir, si se acepta un 0 no podrá volver a aceptarse un 0 hasta que se haya aceptado un 1, y viceversa, si se acepta un 1 no podrá volver a aceptarse un 1 hasta que se haya aceptado un 0.
- El número total de dígitos 0 o 1 aceptados en un instante no puede superar el doble de número de dígitos 2 ya aceptados en dicho instante.

Cuando un productor envía un dígito que no se puede aceptar por el impresor, el productor quedará bloqueado esperando completar el `s_send`. El pseudocódigo de los procesos productores (**Productor**) se muestra a continuación, asumiendo que se usan operaciones bloqueantes no buferizadas (síncronas):

```

1 process Productor[ i : 0,1,2 ];
  begin
    while true do begin
      s_send(i, Impresor);
5    end
  end
end

```

Figura 5.25: Código para el Ejercicio 5.3.8.

Escribir en pseudocódigo el código del proceso **Impresor**, utilizando para ello un bucle infinito con una orden de espera selectiva `select` que permita implementar la sincronización requerida entre los procesos, según el esquema anterior.

El código solicitado es el siguiente, donde `ult0` indica:

- `true` si el último dígito 0 o 1 recibido fue un 0.
- `false` si el último dígito 0 o 1 recibido fue un 1.

Además, notemos que el ejercicio no impone restricciones sobre cuando se pueden recibir dígitos 2.

```

1  process Impresor
   var cant0, cant1, cant2 : integer := 0, 0, 0;
     ult0 : boolean := true;
     n : 0..2;
5  begin
     while true do begin
       select
         when (cant0 < 2*cant2-1 and not ult0) receive(n, Productor[0]) do
           cant0 := cant0 + 1;
10          ult0 := true;
           end

         when (cant1 < 2*cant2-1 and ult0) receive(n, Productor[1]) do
           cant1 := cant1 + 1;
15          ult0 := false;
           end

         when receive(n, Productor[2]) do
           cant2 := cant2 + 1;
20          end
         end

       imprime(n);
     end
25 end

```

Ejercicio 5.3.9. En un sistema distribuido hay un vector de n procesos iguales que envían con **send** (en un bucle infinito) valores enteros a un proceso receptor, que los imprime. Si en algún momento no hay ningún mensaje pendiente de recibir en el receptor, este proceso debe de imprimir “no hay mensajes, duermo”; después de bloquearse durante 10 segundos (con **sleep_for(10)**), antes de volver a comprobar si hay mensajes (esto podría hacerse para ahorrar energía, ya que el procesamiento de mensajes se hace en ráfagas separadas por 10 segundos). Este problema no se puede solucionar usando **receive** o **i_receive**. Indica a que se debe esto. Sin embargo, sí se puede hacer con **select**. Diseña una solución a este problema con **select**:

```

1  process Emisor[ i : 1..n ];
   var dato : integer;
   begin
     while true do begin
5      dato := Producir();
       send(dato, Receptor);
     end
   end
end

```

Figura 5.26: Código para el Ejercicio 5.3.9.

El problema no puede resolverse con instrucciones **receive** o **i_receive** porque:

- En el caso de la instrucción **receive**, si no hay mensajes pendientes, el proceso se bloquearía hasta recibir el primero, pero este no es el comportamiento deseado.
- En el caso de la instrucción **i_receive**, comenzaría instantáneamente la recepción del mensaje, pero el proceso volvería inmediatamente antes de recibirlo, por lo que no sabríamos si hay o no un mensaje pendiente.

Sin embargo, podemos hacer uso de las sentencias **else** de las instrucciones **select**, de forma que una instrucción se ejecute cuando todas las guardas no son ejecutables:

```

1  process Receptor
   var dato : integer;
   begin
       while true do begin
5      select
         for i:= 1 to n
           when receive(dato, Emisor[i]) do
             imprime(dato);
           end
10     else begin
             imprime("No hay mensajes, duermo");
             sleep_for(10);
           end
       end select
15    end
   end

```

Ejercicio 5.3.10. En un sistema tenemos N procesos emisores que envían de forma segura un único mensaje cada uno de ellos a un proceso receptor, mensaje que contiene un entero con el número de proceso emisor. El proceso receptor debe de imprimir el número del proceso emisor que inició el envío en primer lugar. Dicho emisor debe terminar, y el resto quedarse bloqueados:

```

1  process Emisor[ i : 1..N ];
   begin
       s_send(i, Receptor);
   end
5
   process Receptor;
   var ganador : integer;
   begin
       { calcular ganador }
10    ...
       print "El primer envio lo ha realizado: ", ganador;
   end

```

Figura 5.27: Código para el Ejercicio 5.3.10.

Para cada uno de los siguientes casos, describir razonadamente si es posible diseñar una solución a este problema o no lo es. En caso afirmativo, escribe una posible solución:

- (a) el proceso receptor usa exclusivamente recepción mediante una o varias llamadas a `receive`.
- (b) el proceso receptor usa exclusivamente recepción mediante una o varias llamadas a `i_receive`.
- (c) el proceso receptor usa exclusivamente recepción mediante una o varias instrucciones `select`.

Distinguimos casos:

- (a) No es posible, ya que si estamos pensando en usar una única instrucción `receive` de forma que el proceso ganador sea aquel que realice una cita con esta instrucción, entonces no nos quedaríamos con el proceso emisor que inició el envío en primer lugar, sino con el emisor del mensaje que primero llegó en el receptor.
- (b) Tampoco es posible, porque ahora mantenemos el problema anterior pero además el orden en el que se reciben los mensajes en el receptor no tiene por qué coincidir con el orden con el que se realizan las instrucciones `i_receive`.
- (c) Sí que es posible, ya que en caso de que haya más de un mensaje iniciado y preparado para ser recibido, la orden `select` escogerá aquel mensaje cuyo emisor comenzó antes la operación de envío:

```

1  process Receptor
   var ganador : integer;
   begin
     select
5      for i := 1 to N; when receive(ganador, Emisor[i]) do
         null;
       end
     end select
     print "El primer envio lo ha realizado: ", ganador;
10  end

```

Ejercicio 5.3.11. Supongamos que tenemos N procesos concurrentes semejantes. Cada proceso produce $N-1$ caracteres (con $N-1$ llamadas a la función `ProduceCaracter`) y envía cada carácter a los otros $N-1$ procesos. Además, cada proceso debe imprimir todos los caracteres recibidos de los otros procesos (el orden en el que se escriben es indiferente).

- Describe razonadamente si es o no posible hacer esto usando exclusivamente `s_send` para los envíos. En caso afirmativo, escribe una solución.
- Escribe una solución usando `send` y `receive`.

Distinguimos casos:

- En el primer caso, es imposible implementar esta funcionalidad usando operaciones `s_send` de envío síncrono, ya que todos los procesos ejecutarían dicha

instrucción, llevando a un interbloqueo de todos los procesos, situación que se pone de manifiesto de forma simple si simplemente consideramos dos procesos:

```

1 process P1;
  var n1, n2 : integer := 1;
  begin
    send(n1, P2);
5   receive(n2, P2);
  end

```

```

1 process P2;
  var n1, n2 : integer := 2;
  begin
    send(n2, P1);
5   receive(n1, P1);
  end

```

Figura 5.28: Situación típica de interbloqueo.

- Con instrucciones `send` y `receive` sí que se puede resolver de forma sencilla:

```

1 process P[ i : 0..N ];
  var n : char;
  begin
    { Enviamos todos los caracteres }
5   for j := 0 to N do
      if i <> j then begin
        n := Producir();
        send(n, P[j]);
      end
10  end do
    { Recibimos todos los caracteres }
    for j := 0 to N do
      if i <> j then begin
        receive(n, P[j]);
15     print(n);
      end
    end do
  end

```

Ejercicio 5.3.12. Escribe una nueva solución al Ejercicio 5.3.11 en la cual se garantice que el orden en el que se imprimen los caracteres es el mismo orden en el que se inician los envíos de dichos caracteres (pista: usa `select` para recibir).

Como bien indica la pista, basta con usar una instrucción `select` para realizar la recepción de los caracteres:

```

1 process P[ i : 0..N ];
  var n : char;
  begin
    { Enviamos todos los caracteres }
5   for j := 0 to N do
      if i <> j then begin
        n := Producir();
        send(n, P[j]);
      end

```



```

10  end do

    { Recibimos todos los caracteres }
    for j := 1 to N do
        select
15      for k := 0 to N
            when k <> i receive(n, P[k])
                print(n);
            end
        end select
20  end do

```

Ejercicio 5.3.13. Supongamos de nuevo el problema anterior en el cual todos los procesos envían a todos. Ahora cada item de datos a producir y transmitir es un bloque de bytes con muchos valores (por ejemplo, es una imagen que puede tener varios megabytes de tamaño). Se dispone del tipo de datos `TipoBloque` para ello, y el procedimiento `ProducirBloque`, de forma que si `b` es una variable de tipo `TipoBloque`, entonces la llamada a `ProducirBloque(b)` produce y escribe una secuencia de bytes en `b`. En lugar de imprimir los datos, se deben consumir con una llamada a `ConsumirBloque(b)`.

Cada proceso se ejecuta en un ordenador, y se garantiza que hay la suficiente memoria en ese ordenador como para contener simultáneamente, al menos, hasta N bloques. Sin embargo, el sistema de paso de mensajes (SPM) podría no tener memoria suficiente como para contener los $(N - 1)^2$ mensajes en tránsito simultáneos que podría llegar a haber en un momento dado con la solución anterior.

En estas condiciones, si el SPM agota la memoria, debe retrasar los `send` dejando bloqueados los procesos y, en esas circunstancias, se podría producir interbloqueo. Para evitarlo, se pueden usar operaciones inseguras de envío, `i_send`. Escribe dicha solución, usando como orden de recepción el mismo que en el problema anterior.

Para resolver el problema, utilizaremos un array de $N - 1$ bloques a enviar (como vamos a usar la operación `i_send` es un envío inseguro, luego optamos por no modificar los bloques tras enviarlos) y de un bloque para recibir. Además, antes de terminar un proceso tendremos que esperar a que este proceso haya terminado el envío de todos sus bloques:

```

1  process P[ i : 0..N ];
    var bloque : array[0..N] of TipoBloque;
        estado : array[0..N] of Estado;
    begin
2  { 1. Realizamos todos los envíos }
    for j := 0 to N do
        if i <> j then begin
            ProducirBloque(bloque[j]);
            i_send(bloque[j], P[j], estado[j]);
10      end
        end
    end

    { 2. Procesamos las recepciones }
    for j := 0 to N
15      if i <> j then begin

```

```

        receive(bloque[i], P[j]);
        ConsumirBloque(bloque[i]);
    end
end
20
    { 3. Esperamos en caso de que no se hayan realizado }
    { todos los envíos antes de terminar }
    for j := 0 to N
        if i <> j then begin
25            wait_send(estado[j]);
        end
    end
end
end

```

Ejercicio 5.3.14. En los tres problemas anteriores, cada proceso va esperando a recibir un ítem de datos de cada uno de los otros procesos, consume dicho ítem, y después pasa a recibir del siguiente emisor (en distintos órdenes). Esto implica que un envío ya iniciado, pero pendiente, no puede completarse hasta que el receptor no haya consumido los anteriores bloques, es decir, se podría estar consumiendo mucha memoria en el SPM por mensajes en tránsito pendientes cuya recepción se ve retrasada. Escribe una solución en la cual cada proceso inicia sus envíos y recepciones y después espera a que se completen todas las recepciones antes de iniciar el primer consumo de un bloque recibido. De esta forma todos los mensajes pueden transferirse potencialmente de forma simultánea. Se debe intentar que la transmisión y la producción de bloques sean lo más simultáneas posible. Suponer que cada proceso puede almacenar como mínimo $2 \cdot N$ bloques en su memoria local, y que el orden de recepción o de consumo de los bloques es indiferente.

La solución a este último problema es similar a la del Ejercicio 5.3.13, pero en este caso debemos usar la instrucción `i_receive` para recibir, así como un array entero de bloques para realizar dicha recepción. El código sería el siguiente:

```

1  process P[ i : 0..N ];
    var bloque_env : array[0..N] of TipoBloque;
        bloque_rec : array[0..N] of TipoBloque;
        estado_env : array[0..N] of Estado;
5    estado_rec : array[0..N] of Estado;
    begin
        { Inicializamos las recepciones }
        for j := 0 to N do
            if j <> i then begin
10                i_receive(bloque_rec[j], P[j], estado_rec[j]);
            end
        end

        { Inicializamos los envíos }
15        for j := 0 to N do
            if j <> i then begin
                ProducirBloque(j);
                i_send(bloque_env[j], P[j], estado_env[j]);
            end
20        end
    end

```

```
    { Esperar a que terminen todas las recepciones }  
    for j := 0 to N do  
        if j <> i then begin  
25         wait_rcv(estado_rec[j]);  
        end  
    end  
  
    { Procesar todos los bloques }  
30    for j := 0 to N do  
        if j <> i then begin  
            ConsumirBloque(bloque_rcv[j]);  
        end  
    end  
  
35    { Esperar a que terminen todos los envíos antes de terminar el proceso }  
    for j := 0 to N do  
        if j <> i then begin  
40         wait_send(estado_env[j]);  
        end  
    end  
end
```

5.4. Sistemas de Tiempo Real

Ejercicio 5.4.1. Dado el conjunto de tareas periódicas y sus atributos temporales que se indica en la Tabla 5.8, determinar si se puede planificar el conjunto de dichas tareas utilizando un esquema de planificación basado en planificación cíclica. Diseña el plan cíclico determinando el marco secundario, y el entrelazamiento de las tareas sobre un cronograma.

Tarea	C_i	T_i	D_i
T1	10	40	40
T2	18	50	50
T3	10	200	200
T4	20	200	200

Tabla 5.8: Tareas periódicas y sus atributos temporales.

Ejercicio 5.4.2. El siguiente conjunto de tareas periódicas se puede planificar con ejecutivos cíclicos. Determina si esto es cierto calculando el marco secundario que debería tener. Dibuja el cronograma que muestre las ocurrencias de cada tarea y su entrelazamiento. ¿Cómo se tendría que implementar? (escribe el pseudo-código de la implementación)

Tarea	C_i	T_i	D_i
T1	2	6	6
T2	2	8	8
T3	3	12	12

Tabla 5.9: Tareas periódicas y sus atributos temporales.

Ejercicio 5.4.3. Comprobar si el conjunto de procesos periódicos que se muestra en la siguiente tabla es planificable con el algoritmo RMS utilizando el test basado en el factor de utilización del tiempo del procesador. Si el test no se cumple, ¿debemos descartar que el sistema sea planificable?

Tarea	C_i	T_i
T1	9	30
T2	10	40
T3	10	50

Tabla 5.10: Tareas periódicas y sus atributos temporales.

Ejercicio 5.4.4. Considérese el siguiente conjunto de tareas compuesto por tres tareas periódicas:

Tarea	C_i	T_i
T1	10	40
T2	20	60
T3	20	80

Tabla 5.11: Tareas periódicas y sus atributos temporales.

Comprueba la planificabilidad del conjunto de tareas con el algoritmo RMS utilizando el test basado en el factor de utilización. Calcular el hiperperiodo y construir el correspondiente cronograma.

Ejercicio 5.4.5. Comprobar la planificabilidad y construir el cronograma de acuerdo al algoritmo de planificación RMS del siguiente conjunto de tareas periódicas.

Tarea	C_i	T_i
T1	20	60
T2	20	80
T3	20	120

Tabla 5.12: Tareas periódicas y sus atributos temporales.

Ejercicio 5.4.6. Determinar si el siguiente conjunto de tareas puede planificarse con la política de planificación RMS y con la política EDF, utilizando los tests de planificabilidad adecuados para cada uno de los dos casos. Comprobar también la planificabilidad en ambos casos construyendo los dos cronogramas.

Tarea	C_i	T_i
T1	1	5
T2	1	10
T3	2	20
T4	10	20
T5	7	100

Tabla 5.13: Tareas periódicas y sus atributos temporales.

Ejercicio 5.4.7. Describe razonadamente si el siguiente conjunto de tareas puede planificarse o no puede planificarse en un sistema monoprocesador usando un ejecutivo cíclico o usando algún algoritmo basado en prioridades estáticas o dinámicas.

Tarea	C_i	T_i
T1	1	5
T2	1	10
T3	2	10
T4	11	20
T5	5	100

Tabla 5.14: Tareas periódicas y sus atributos temporales.

Problemas adicionales

Ejercicio 5.4.8. Para el conjunto de tareas cuyos datos se muestran más abajo, se pide:

- Dibujar el gráfico de ejecución y obtener el tiempo de respuesta de cada tarea.
- Determinar, mediante inspección del gráfico anterior, cuántas veces interfiere la tarea \mathcal{T}_1 a la tarea \mathcal{T}_3 durante un intervalo temporal dado por el tiempo de respuesta de esta última tarea.
- Hacer lo mismo que en el apartado anterior pero para las tareas \mathcal{T}_1 y \mathcal{T}_2 .

Tarea	C_i	T_i	D_i
T1	1	3	2
T2	3	6	5
T3	2	13	13

Tabla 5.15: Tareas periódicas y sus atributos temporales.

Ejercicio 5.4.9. Verificar la planificabilidad del siguiente conjunto de tareas utilizando para ello el algoritmo del “primero el del tiempo límite más cercano” (EDF).

Tarea	C_i	T_i
T1	1	4
T2	2	6
T3	3	8

Tabla 5.16: Tareas periódicas y sus atributos temporales.

Ejercicio 5.4.10. Verificar la planificabilidad utilizando el algoritmo EDF de asignación dinámica de prioridades a las tareas y construir el diagrama de ejecución de tareas del siguiente conjunto:

Tarea	C_i	D_i	T_i
T1	2	5	6
T2	2	4	8
T3	4	8	12

Tabla 5.17: Tareas periódicas y sus atributos temporales.

Ejercicio 5.4.11. Verificar la planificabilidad del conjunto de tareas descrito en el Ejercicio 5.4.10 utilizando el algoritmo del “plazo de respuesta máximo (D)” (algoritmo *deadline monotonic* o DM).

Ejercicio 5.4.12. Indicar cuáles de las siguientes afirmaciones son correctas respecto de los algoritmos para resolver el problema de *inversión de prioridad* de las tareas en sistemas de tiempo real de *misión crítica*:

- (a) Suponiendo que las tareas se planifican con el protocolo de *herencia de prioridad*: la prioridad heredada por una tarea sólo se mantiene mientras dicha tarea esté utilizando un recurso compartido con otra tarea más prioritaria.
- (b) Con el protocolo de *techo de prioridad*, cuando una tarea adquiere un recurso no puede verse interrumpida, hasta que termine su ejecución, por otras tareas que se activan después que ésta y que vayan a utilizar en el futuro un recurso con límite de prioridad igual o inferior.
- (c) Con el protocolo de techo de prioridad (PPP), una tarea no puede comenzar a ejecutarse si no están libres todos los recursos que va a utilizar durante su primer ciclo.
- (d) Si consideramos una tarea periódica que utilice el protocolo de techo de prioridad inmediato para cambiar su prioridad dinámica cuando accede a recursos, siempre se cumplirá que dicha tarea no puede ser interrumpida por otra menos prioritaria que ella.
- (e) Con el protocolo de techo de prioridad las tareas más prioritarias del sistema pueden ser interrumpidas durante cada ciclo de su ejecución como máximo 1 vez cuando acceden a recursos que comparten con otras tareas menos prioritarias.
- (f) El protocolo de techo de prioridad original producirá siempre tiempos de respuesta menores para las tareas que el algoritmo de *herencia de prioridad*.

Ejercicio 5.4.13. Calcular la utilización máxima del procesador que se puede asignar al *Servidor Esporádico* para garantizar la planificabilidad del siguiente conjunto de tareas periódicas utilizando RM.

\mathcal{T}_1	1	5
\mathcal{T}_2	2	8

Tabla 5.18: Conjunto de tareas periódicas.

Ejercicio 5.4.14. Calcular la utilización máxima del procesador que puede ser asignada al *Servidor Diferido* (SD) para garantizar la planificabilidad del conjunto de tareas periódicas dado en el Ejercicio 5.4.13 anterior.

Ejercicio 5.4.15. Junto con las tareas periódicas que se muestran en el Ejercicio 5.4.13 definir un plan para planificar las siguientes tareas aperiódicas utilizando un *SD* (tarea sondeante) que posea una utilización máxima del tiempo del procesador y prioridad intermedia:

	t_a	C_i
J_1	2	2
J_2	7	2
J_3	17	1

Tabla 5.19: Tareas periódicas y sus atributos temporales.

Ejercicio 5.4.16. Resolver ahora el mismo problema de planificación descrito en el Ejercicio 5.4.15 utilizando ahora un *Servidor Esporádico* que tenga una utilización máxima y prioridad intermedia.

Ejercicio 5.4.17. Resolver el mismo problema de planificación descrito en el Ejercicio 5.4.15 utilizando ahora un *Servidor Diferido* que tenga una utilización máxima y prioridad intermedia.

Ejercicio 5.4.18. Utilizar un *Servidor Esporádico* con capacidad $C_s = 2$ y periodo $T_s = 6$ para planificar las siguiente tareas:

	C_i	T_i
\mathcal{T}_1	1	4
\mathcal{T}_2	2	6
	a_i	C_i
J_1	2	2
J_2	5	1
J_3	10	2

Tabla 5.20: Tareas periódicas y sus atributos temporales.