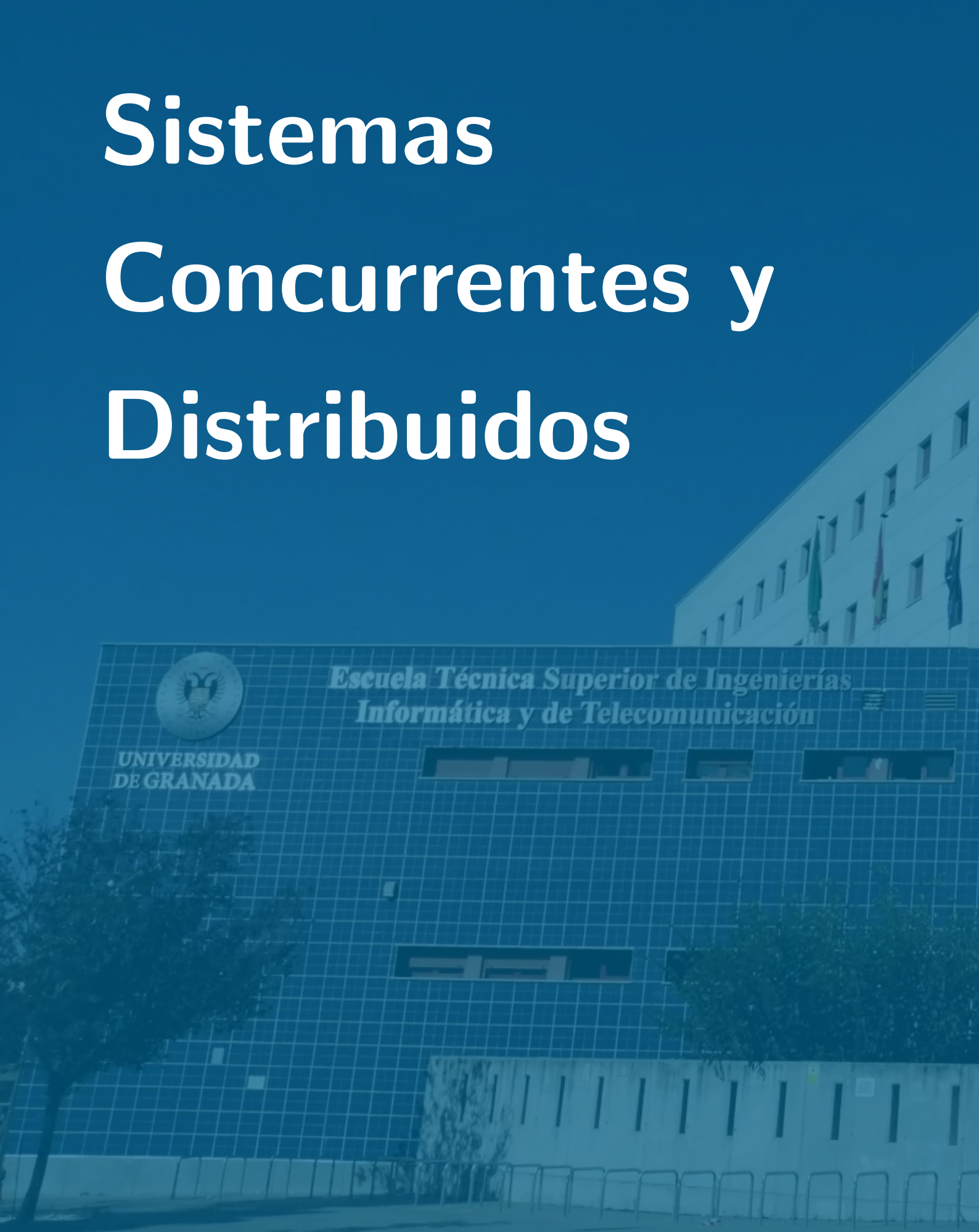
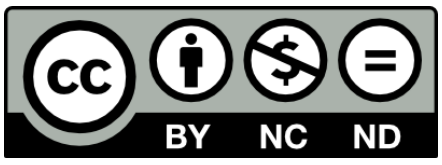
Sistemas Concurrentes y Distribuidos



Los Del DGIIM, losdeldgiim.github.io

Doble Grado en Ingeniería Informática y Matemáticas
Universidad de Granada



Sistemas Concurrentes y Distribuidos

Los Del DGIIM, `losdeldgiim.github.io`

José Juan Urrutia Milán
Arturo Olivares Martos

Granada, 2024-2025

Índice general

1. Sincronización en memoria compartida. Monitores

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

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

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

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

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

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

1.1. Definición de un monitor

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

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

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

que interrumpir la ejecución del procedimiento.

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

Ventajas

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

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

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

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

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

1.1.1. Concepto de monitor

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

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

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

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

Variables permanentes
Procedimientos exportados
Código de inicialización

Tabla 1.1: Esquema de un monitor.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.1.2. Características de programación con monitores

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

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

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

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

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

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

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

[7]Se explicará más adelante.

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

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

Instanciación de monitores

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

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

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

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

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

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

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

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

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

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

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

Exclusión mutua en los procedimientos de un monitor

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

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

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

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

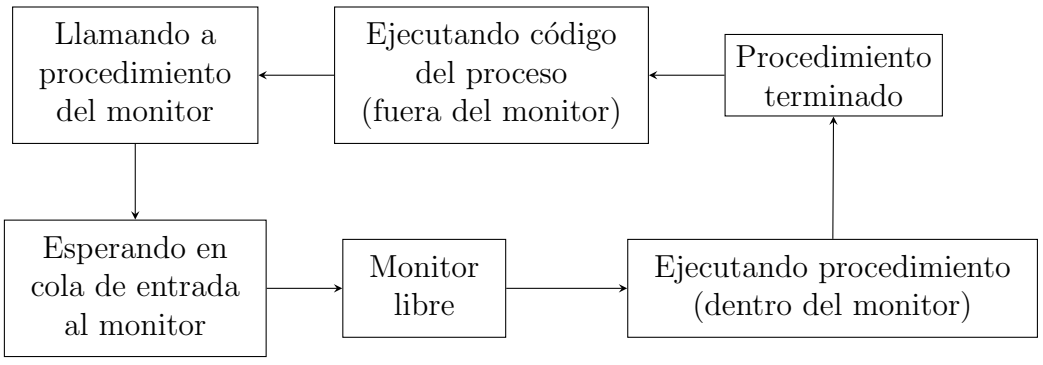


Figura 1.1: Vida de un proceso en un programa concurrente con monitores.

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

Cola del monitor
Variables permanentes
Procedimientos exportados
Código de inicialización

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

1.1.3. Operaciones de sincronización

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

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

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

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

Tipo de dato `cond`

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Semántica desplazante

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

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

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

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

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

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

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

1.2. Verificación de programas con monitores

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

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

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

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

1.2.1. Invariante de monitor

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

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

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

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

Esquema de demostración

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

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

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

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

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

por cada procedimiento que tenga nuestro monitor.

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

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

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

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

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

Sin embargo, en caso de ejecutar el siguiente código:

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

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

1.2.2. Axiomas para operaciones de sincronización con semántica desplazante

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

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

para cualesquiera asertos P y Q.

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

Axioma de operación `wait`

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

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

Resumiendo:

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

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

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

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

Axioma de operación `signal`

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

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

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

o equivalentemente:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Para tener probado el triple del procedimiento `P`.

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

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

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

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

Basta con probar los triples

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

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

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

□

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

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

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

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

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

     procedure V;
```

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

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

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

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

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

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

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

Obteniendo así la verificación de nuestro programa concurrente.

2. Relaciones de problemas