Capítulo 1

Introducción a la Programación Concurrente

1.1 Conceptos básicos y motivación

Si un programa secuencial es un conjunto de declaraciones de datos e instrucciones que se ejecutan siguiendo una sola secuencia de ejecución, entonces podemos entender un *programa concurrente* como aquel código que especifica dos ó más *procesos* que *cooperan* en la realización de una determinada tarea.

El concepto de proceso ha de ser entendido como una entidad software abstracta, dinámica, activa, que ejecuta instrucciones y alcanza diferentes estados[1]. Hay que tener presente que el conjunto de instrucciones de un programa, solamente, *no* es un proceso. Por eso resulta engañosa la definición de proceso que hacen los antiguos textos de sistemas operativos como "un programa secuencial en ejecución". Ya que, para que el procesador obtenga toda la información asociada a un proceso en cada momento de su ejecución es necesario también que conozca la información asociada a su *estado*. Desde un punto de vista abstracto, en dicho estado se incluyen:

- los valores de los registros del procesador –el más importante es el valor del contador de programa (*pc*),
- conjuntos de valores de variables propias ubicadas en diferentes tipos de memoria –se corresponden con estados de la pila (*sp*) y del heap,
- acceso a los dispositivos, ficheros, etc., de los que dicho proceso es "propietario" y que normalmente *protege* de un acceso incontrolado por parte de los otros procesos.

Desde el punto de vista del sistema operativo (ver figura 1.1), un proceso se caracteriza por:

[1] por ahora, entendemos el concepto de estado como el conjunto de valores de variables "visibles" y no visibles (pc, sp) en cada instante de la ejecución

1. Zona de memoria, estructurada en varias secciones:
   - *textual*: incluye la secuencia de instrucciones que se están ejecutando,
   - *datos*: espacio de tamaño fijo ocupado por variables globales (o *estáticas*),
   - *pila*: espacio de tamaño variable ocupado por variables locales –su ámbito y duración coinciden con la del proceso,
   - *memoria dinámica o heap*: espacio ocupado por variables dinámicas –normalmente asociadas a punteros, asignadas y destruidas antes de la terminación del proceso.
2. Variables *internas* que cada proceso tiene asociadas de forma implícita:
   - *contador de programa (pc)*: es una dirección en memoria, que se encuentra en la sección de texto, y que contiene la siguiente instrucción a ejecutar por el proceso,
   - *puntero de pila (sp)*: es una dirección en memoria de la zona de pila que indica la última posición ocupada por la pila.

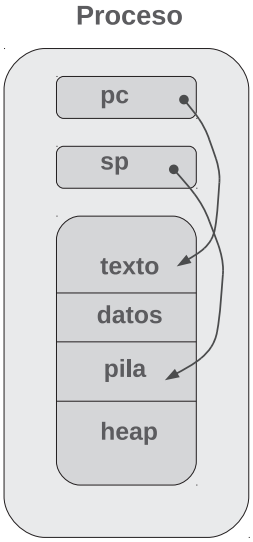

Figura 1.1: Representación de un proceso

En un programa concurrente existirán muchas secuencias de ejecución de instrucciones con, al menos, un flujo o *hebra* de control de ejecución independiente por cada uno de los procesos que lo componen. Si la plataforma de ejecución de nuestro programa sólo tiene un procesador, los múltiples flujos de control se entremezclan y forman uno solo. Esto lo entendemos como una ejecución "pseudo–paralela", independientemente del número de procesadores que tenga la plataforma sobre la que se ejecuta el programa.

Un programa concurrente será en general más eficiente que un programa secuencial, ya que permite a los procesos que hacen E/S no monopolizar el procesador. Dichos procesos se quedarían bloqueados, esperando datos, mientras los otros procesos pueden conseguir el procesador y seguir ejecutándose. Por consiguiente, para procesos destinados a interaccionar con su entorno, entradas/salidas frecuentes, recepción de mensajes, excepciones, señales, etc., el aprovechamiento del tiempo del procesador que se consigue con un programa concurrente es mejor que con uno secuencial programado para realizar la misma tarea o conseguir los mismos resultados. Por otra parte, un programa concurrente siempre modelará mejor a un sistema real que un programa secuencial, ya que los sistemas reales suelen estar compuestos de varias

actividades que se ejecutan en paralelo y cada una de ellas puede ser modelada de forma natural mediante un proceso independiente que se ejecute concurrentemente con el resto.

Ejemplo 1: Un sistema de transaciones para reservas de vuelos. Cada uno de los procesos concurrentes del programa representaría a cada una de las terminales donde se hacen reservas, anulaciones, emisión de billetes, etc.

Ejemplo 2: En un sistema operativo sencillo, cada instancia de programa *shell* (csh, bash, sh, ...) puede estar ejecutándose independientemente si se crea un proceso concurrente para ejecutarla.

En consecuencia, los términos *concurrente*, *concurrencia*, etc., sirven para describir el potencial de ejecución paralela que posee un programa, aplicación o sistema. Podemos entonces definir la Programación Concurrente de una forma más concreta como: *el conjunto de notaciones y técnicas de programación utilizadas para expresar el paralelismo potencial de los programas y para resolver los problemas de sincronización y comunicación que se presenten entre los procesos.*

1.1.1 Modelo abstracto de la Programación Concurrente

La Programación Concurrente (PC) no es sólo un *estilo* o *paradigma* de programación, sino que es, sobretodo, un *modelo abstracto de computación* que sirve para expresar a un nivel de abstracción adecuado [2] el paralelismo potencial de los programas, independientemente de la implementación del paralelismo a nivel de arquitectura del sistema.

Los buenos lenguajes concurrentes deben de proporcionar primitivas de programación para resolver problemas de sincronización y comunicación que sean utilizables en diferentes arquitecturas de máquinas, pero... ¡Esto es un ideal! El modelo abstracto que vamos a introducir consigue, sin embargo, avanzar mucho en esta dirección para alcanzar los siguientes objetivos fundamentales:

- Poder razonar la solución de los problemas a un nivel adecuado, sin entrar en detalles de demasiado bajo nivel que impidan obtener dichas soluciones de una forma sencilla.
- Facilitar la notación, ya que se pueden utilizar lenguajes de programación, y primitivas de comunicación y sincronización de alto nivel, sin necesidad de utilizar código de bajo nivel o llamadas al sistema para resolver los problemas que nos interesan.
- Los programas desarrollados son independientes de una máquina concreta y por tanto, dichos programas son *transportables*.

Es decir, gracias a la definición del modelo anterior, que han de cumplir todos los lenguajes modernos de PC, podemos afirmar que un programa concurrente se podrá ejecutar en diferentes plataformas cumpliendo las mismas propiedades fundamentales[3] de corrección.

[2] es decir, un nivel de astracción que nos permita resolver problemas que nos interesan sin atascarnos en los detalles de bajo nivel

[3] se garantizan, al menos, las propiedades concurrentes de *seguridad* y *vivacidad*

Hipótesis del modelo abstracto de la Programación Concurrente

1. *Atomicidad y entrelazamiento de las instrucciones de los procesos*: a partir de un programa escrito en un lenguaje de alto nivel siempre se puede llegar a un conjunto de instrucciones equivalentes generadas en el nivel más básico posible.

```
  P1          P2          secuencias posibles
--------    ---------    ---------------------
  I11         I21          I11  I11   I11 . . .
  I12         I22          I21  I12   I21 . . .
   .           .           I12  I21   I22 . . .
   .           .           I22  I13   I23 . . .
   .           .           I13  I22   I12 . . .
                            .    .      .
```

Figura 1.2: Secuencias de entrelazamiento de instrucciones atómicas de 2 procesos

   Dicho nivel de instrucciones se corresponde, normalmente, con el repertorio de instrucciones a nivel máquina o ensamblador de los sistemas. Y sus instrucciones se ejecutan sin ser interrumpidas por un cambio de contexto o cualquier otra interrupción del sistema, es decir, las instrucciones se ejecutan de una forma indivisible u *atómica*.

   El que los procesos de un programa concurrente se ejecuten utilizando paralelismo real o no, no influye en los resultados del programa, sólo en la rapidez con se obtienen estos. Según la interpretación anterior, en la figura 1.2 se tiene un programa P que contiene 2 procesos concurrentes $\{P_1, P_2\}$. El conjunto de todas las secuencias posibles de *entrelazamiento* de instrucciones atómicas de P_1 y de P_2 constituyen todas las *secuencias de ejecución* observables o *comportamiento* del programa P. El resultado de la ejecución de P será una de estas secuencias que pertenecen a su *comportamiento*[4] observable, pero no podemos saber cuál ni tampoco influir para que sea una de ellas. A esto se le denomina *no determinismo* en la ejecución de los programas concurrentes. Esta interpretación es un modelo de paralelismo independiente de la arquitectura concreta que posea la plataforma sobre la que se ejecute el citado programa.

2. *Coherencia de acceso concurrente a los datos*: la ejecución concurrente de 2 instrucciones atómicas que acceden a una misma dirección en memoria ha de producir el mismo resultado tanto si dichas instrucciones se ejecutan con *paralelismo real*[5] o con *pseudoparalelismo*, es decir, secuencialmente –una después de otra en un orden arbitrario. Al terminar de acceder, los procesos han de dejar la memoria en un estado coherente con el tipo al que pertenece la variable. En la figura 1.3 se puede ver que el resultado será 1 ó 2, pero es consistente con los valores que puede tomar la variable `x` según las asignaciones del programa. Si por el contrario, I1 e I2 accediesen simultáneamente a la posición de memoria de la variable compartida, se podría producir mezcla de datos y el valor final de `x` podría ser arbitrario.

   Esta suposición está soportada por el hardware de los sistemas reales. La coherencia de los datos tras un acceso concurrente está asegurada por el controlador de memoria.

[4] se define como el conjunto de todas las *secuencias de entrelazamiento* de instrucciones atómicas que se generan a partir del código de los procesos

[5] es decir, cada una de ellas sería ejecutada simultáneamente por un procesador diferente

```
P1          P2                          secuencias
I1:x:=1     I2:x:=2      I1: <STORE x, 1>  |  I2: <STORE x, 2>
...         ...          I2: <STORE x, 2>  |  I1: <STORE x, 1>
                              {x=2}               {x=1}
```

Figura 1.3: Secuencialización de los procesos en el acceso a la memoria

3. *Irrepetitibilidad de la secuencia de instrucciones*: el número de secuencias de entrelazamiento de instrucciones atómicas que se pueden formar a partir de un programa es inabarcable. Por lo que es bastante improbable que 2 ejecuciones seguidas de un programa repitan la misma secuencia de instrucciones atómicas. Esto hace muy difícil la depuración y el análisis de corrección de los programas. Ocurren los denominados *errores transitorios*, esto es, errores que aparecen en unas secuencias de ejecución y en otras no. Por consiguiente, es necesario utilizar métodos formales, basados en la Lógica Matemática, para verificar la corrección de los programas concurrentes.

4. *Velocidad de ejecución de los procesos:* la corrección de los programas concurrentes no puede depender de la velocidad de ejecución relativa de unos procesos con respecto a otros. Si no se cumpliese este requisito, se producidirían las siguientes anomalías de ejecución:
   - *Falta de transportabilidad:* si se hicieran suposiciones acerca de la velocidad de ejecución de los procesos para llegar a los resultados esperados, los programas concurrentes dejarían de ser transportables, ya que en otra plataforma de ejecución, con procesadores de características distintas, no funcionarían correctamente.
   - *Condiciones de carrera:* un programa cuya corrección dependa de la velocidad de ejecución de cada proceso está sujeto a errores transitorios.

```
     P1                    P2
a:= datos;            b:= datos;
a++;                  b--;
datos:= a;            datos:= b;
```

Figura 1.4: condición de carrera entre 2 procesos

En la figura 1.4, se puede ver una condición de carrera en el acceso a la variable *datos* (inicialmente = 0) por parte de los procesos `P1` y `P2`. La variable aludida puede terminar con el valor +1, 0, -1, y no hay forma de predecirlo, dependerá de la velocidad y el orden en que los procesos accedan a las variables.

No obstante, hay una excepción para esta hipótesis del modelo. Existe un tipo de programas, que normalmente son concurrentes, en los que sí se hace suposición acerca del tiempo de ejecución de determinados procesos: son los programas para sistemas de *tiempo real*.

Por último, dentro del modelo abstracto de la Programación Concurrente se asume la *Hipótesis de progreso finito*, que está relacionada con la velocidad de ejecución de los procesos. Sin esta hipótesis, para cualquier plataforma donde se pueda ejecutar, no se podría asegurar nada sobre la satisfacción de las propiedades de corrección de un programa concurrente. Por ejemplo, propiedades como *vivacidad* de los procesos no podrían llegar a demostrarse.

Hipótesis de progreso finito de los procesos: todos los procesos de un programa concurrente conseguirán alguna vez ejecutarse. Esto se puede entender a 2 niveles:

- *Globalmente:* si existe al menos 1 proceso preparado, eventualmente se permitirá la ejecución de algún proceso del programa.
- *Localmente:* si un proceso comienza la ejecución de una sección de código, eventualmente completará su ejecución.

En resumen, la hipótesis de *progreso finito* se podría enunciar de esta manera: *no tenemos ninguna razón para suponer que un proceso detendrá su ejecución injustificadamente.*

1.1.2 Consideraciones sobre el hardware

Atendiendo a la arquitectura, los sistemas implementan la concurrencia según 3 modelos, como se puede ver en la figura 1.5:

- *Sistemas monoprocesador:* Los procesos se ejecutan compartiendo el tiempo de un único procesador. Se modeliza la concurrencia mediante el entrelazamiento de las instrucciones de los procesos.
- *Sistemas multiprocesador:* Existe más de un procesador que puede ejecutar simultáneamente instrucciones. Existe una memoria común a través de la cual se pueden comunicar los procesos.
- *Sistemas distribuidos:* Cada procesador tiene una memoria independiente. Para poder comunicar a los procesos hay que utilizar una red de comunicaciones. Se utiliza el paso de mensajes para implementar tanto la comunicación como la sincronización entre los procesos. Son más complicados de programar que los sistemas anteriores

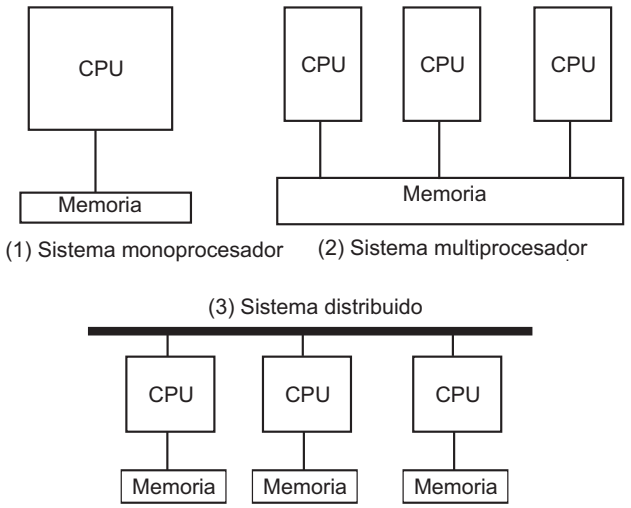

Figura 1.5: Arquitecturas de sistemas con diferentes grados de paralelismo

Se dice que existe *paralelismo real* cuando se tiene un sistema multiprocesador o distribuido. Si se tiene un sistema monoprocesador, entonces se dice que existe *pseudoparalelismo* o *paralelismo virtual*. El tipo de paralelismo afecta a la eficiencia, pero no a la corrección de los programas. De hecho los lenguajes de Programación Concurrente, que se han desarrollado para ser implementados en varios tipos de arquitecturas, permiten transportar los programas de plataformas monoprocesador a otras multiprocesador de una forma transparente.

1.2 Exclusión mutua y sincronización

La exclusión mutua asegura que una serie de sentencias del texto de un proceso, compartidas con otros procesos del programa, se ejecutarán de una forma *atómica* o indivisible. No se permitirá, por tanto, que varios procesos del programa ejecuten un bloque de estas sentencias concurrentemente. A dicho bloque (o sección[6]) de sentencias dentro del texto de un proceso se le denomina *sección crítica*.

Con las primitivas de Programación Concurrente adecuadas, si 2 ó más procesos intentan ejecutar una sección crítica, sólo lo conseguirá uno al tiempo; los demás han de esperar hasta que dicho proceso acabe y entonces lo vuelven a intentar. Un ejemplo de acceso en exclusión mutua es la asignación de recursos (respaldo de archivos, plotters, impresoras, etc.) que hace, por ejemplo, un sistema operativo a los distintos procesos de usuario, ya que si mezclaran los trabajos enviados por estos, la salida producida por el dispositivo sería inaprovechable. La estructura que suele tener el código de un proceso que incluye una sección crítica es:

```
Resto; //operaciones fuera de la seccion critica
Protocolo de adquisicion;
Sentencias que pertenecen a la seccion critica
Protocolo de restitucion;
```

Los protocolos de adquisición y restitución son bloques de instrucciones cuyo objetivo en conjunto es conseguir que se cumpla la condición de exclusión mutua en el acceso de un proceso a la sección crítica. El primer protocolo aludido decide, en el caso de que haya competencia de varios procesos, cuál de ellos entra en sección crítica; al resto no se les permite avanzar hasta que termine de ejecutarla el proceso que entró. El protocolo de restitución sirve para permitir a un nuevo proceso –que posiblemente se quedó esperando al ejecutar el protocolo de adquisición– entrar en sección crítica.

1.2.1 Sincronización

En los programas concurrentes los procesos tienen necesidad de comunicarse para cooperar en la realización de una tarea común. Como antes se ha discutido, la comunicación entre los procesos se puede realizar mediante variables compartidas o mensajes, dependiendo del tipo de plataforma (ver sección 1.1.1) sobre la que se ejecute el programa concurrente. La comunicación entre procesos da lugar a la necesidad de que exista una sincronización entre ellos durante la ejecución.

Existen dos casos fundamentales de sincronización:

- *Sincronización con condiciones:* consiste en detener la ejecución de un proceso hasta que se cumpla una determinada condición. A esto se le llama *sincronizar el proceso con la condición*.
- *Exclusión mutua:* se puede considerar una condición de sincronización particular: un proceso no puede avanzar hasta que la sección crítica quede libre y se le autorice a entrar en ella.

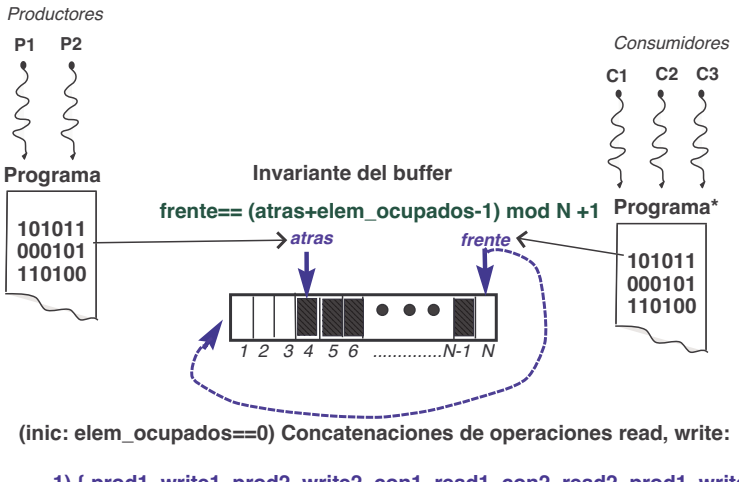

Figura 1.6: Representación del funcionamiento del buffer en un *productor-consumidor*

La programación de un *buffer circular* finito[7] (ver figura 1.6) es un ejemplo típico que ayuda a entender la diferencia entre las 2 formas de sincronización anteriores.

Al buffer circular acceden procesos productores para insertar datos y procesos consumidores para retirarlos. El objetivo de la estructura de datos *buffer* es la de "desacoplar" la ejecución de los procesos de tipo consumidor y productor. Por ejemplo, si inicialmente se ejecutan una tanda de productores, y todavía no hay ningún proceso que pueda consumir los datos producidos, entonces estos se guardan temporalmente en el buffer a la espera de ser retirados por los procesos consumidores cuando se ejecuten. En este ejemplo, la exclusión se utiliza para asegurar que un productor y un consumidor no acceden al buffer al mismo tiempo y, por tanto, se pueda producir una *condición de carrera* en el acceso concurrente a un mismo elemento[8]. La sincronización con condiciones se utilizará para asegurar que un mensaje introducido no sea sobrescrito antes de ser leído, o bien evitar la inserción de datos en un buffer lleno; o un intento de extracción de un bufer vacío, que aparece si se intenta consumir 2 veces después de haber producido e insertado sólo 1 elemento en un buffer vacío, tal como muestra la secuencia etiquetada "ilegal" de la figura 1.6.

Desde el punto de vista del *modelo abstracto*, el papel de la sincronización consiste en restringir el conjunto de todas las posibles secuencias de ejecución a sólo aquellas que pueden considerarse correctas desde el punto de vista de las propiedades que ha de cumplir el programa.

[6]podría no coincidir con un bloque del programa, según se entiende en la mayoría de lenguajes

[7]puede tener la estructura de una *cola de datos circular*

[8]si se diera en un buffer que contuviera elementos complejos, se podría llegar a leer del buffer un mensaje parcialmente escrito

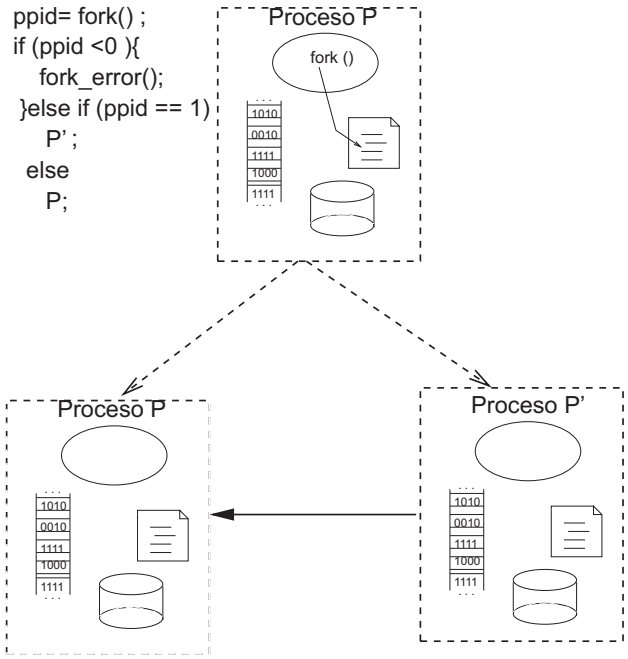

Figura 1.7: Operación fork() en UNIX

1.2.2 Creación de procesos

Las primeras notaciones, que se incluyeron en lenguajes secuenciales para expresar la ejecución concurrente de procesos en un programa, no separaban la creación de los procesos de la sincronización. Los lenguajes y sistemas modernos con facilidades para la programación concurrente separan ambos conceptos e imponen una estructura al programa y a sus procesos.

La primera idea en este sentido fue la *declaración de procesos*. En los programas se declaran explícitamente rutinas que se van a ejecutar concurrentemente. Los mecanismos de sincronización aparecen como otras instrucciones más en el código.

La declaración de procesos puede ser:

- *estática:* se declara un número fijo de procesos que se activan cuando se inicia la ejecución del programa.

- *dinámica:* se especifica un número variable de procesos que se activan en cualquier momento. Con esta modalidad se pueden desactivar procesos que no interesan durante la ejecución de un programa.

Ejemplo de lenguajes que poseen alguno de estos tipos de declaración de procesos son: MPI [Snir et al., 1999] con la definición de grupos de procesos conectados en una sesión[9], Occam [INMOS, 1984] , Pascal Concurrente [Brinch-Hansen, 1975], Modula [Wirth, 1985]. El ejemplo más típico de lenguaje que también incluye creación dinámica de procesos es Ada [Barnes, 1994].

[9]cada "communicator" proporciona a cada proceso contenido en el grupo un identificador independiente en una topología ordenada

Creación mediante ramificación

Es una forma no estructurada de creación de procesos, empleada por los sistemas operativos UNIX y Linux.

Instrucción de ramificación (`fork()`)
La instrucción `fork()` ramifica el flujo de control del programa o proceso que la llama. Como resultado la función llamada puede comenzar a ejecutarse como un proceso concurrente independiente, que entrelazará sus instrucciones con el resto del programa. En la figura 1.7 puede verse que `fork()` produce una segunda copia (P'), totalmente independiente del proceso (P) que llama a dicha función, este último continua ejecutando el resto de sus instrucciones después de hacer la llamada.

Instrucción de unión (`join()`)

La operación de unión permite que un proceso P, mediante una llamada a `join()`, espere a que otro proceso P' termine. P es el proceso que invoca la operación de unión de flujos de control, y P' el proceso objetivo. Al finalizar la llamada a `join()` se puede estar seguro que el proceso objetivo ha terminado con seguridad. La ventaja que tiene este mecanismo –frente a que el proceso P realice espera ociosa hasta que P' haya terminado– es que el proceso que hace la llamada no consume ciclos del procesador durante dicha espera. Si al hacer el proceso P la llamada a la operación `join()` el proceso objetivo hubiera terminado ya, entonces la llamada a dicha operación no produce ningún efecto.

Las ventajas de la ramificación frente a otros mecanismos de creación de procesos consiste en que esta forma de creación es práctica y potente. Permite la creación dinámica, no estructurada, de procesos concurrentes. Los inconvenientes es su falta de estructuración, ya que al poder aparecer en bucles, condicionales, funciones recursivas, etc. hace muy difícil comprender el funcionamiento de los programas, lo que implica dificultades para su verificación y depuración.

La sentencia *cobegin-coend*

Es una sentencia estructurada de creación de procesos. La sentencia COBEGIN inicia la ejecución concurrente de las instrucciones, nombres de procesos, llamadas a procedimentos, etc. que aparezcan a continuación de esta instrucción:

```
COBEGIN S1;S2; ...; Sn COEND;
```

La instrucción siguiente al COEND, en el fragmento de programa anterior, sólo se ejecutará cuando todas las sentencias componentes: S1, S2, ..., Sn hayan terminado.

Se dice que es una sentencia estructurada, ya que el flujo de control del programa cuando llega hasta la palabra COBEGIN tiene un único punto de entrada y un único punto de salida, tras ejecutar COEND. El bloque donde aparece depende de la terminación de la sentencia COBEGIN-COEND para completar su ejecución dentro del programa al que pertenece.

Creación de hebras POSIX 1003

La independencia que existe entre las zonas de memoria propia de los procesos en UNIX proporciona protección implícita en el acceso a variables del programa, pero no es flexible para definir mecanismos de sincronización en interacciones cooperativas, ya que presenta los siguientes inconvenientes:

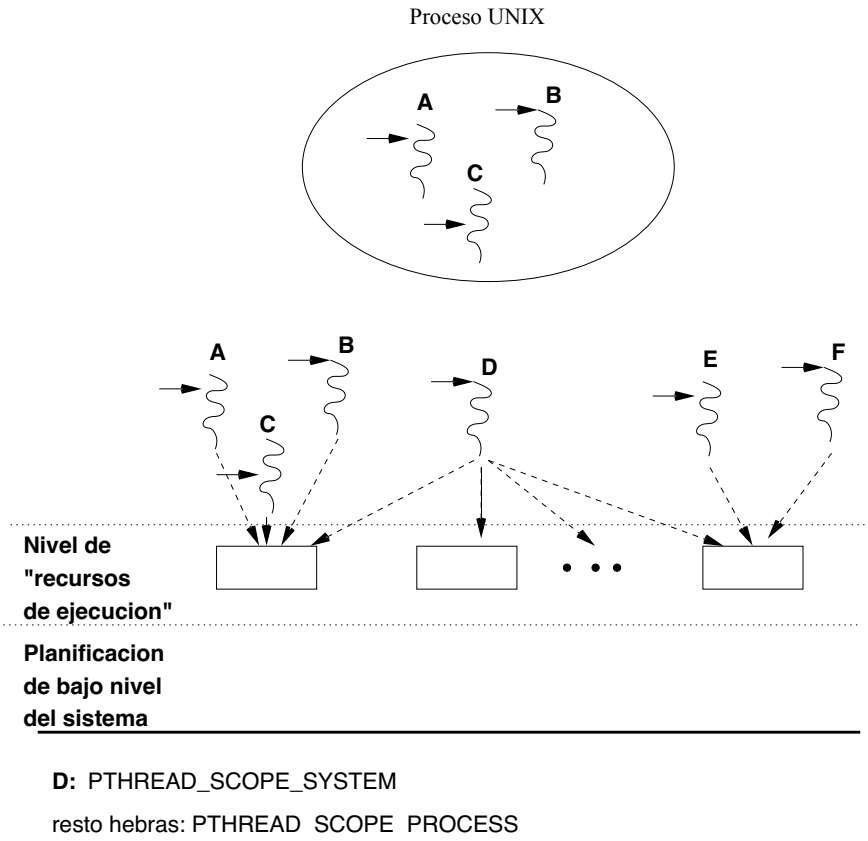

Figura 1.8: Representación de planificación de hebras POSIX 1003.1c (thread package)

- El *coste* del cambio de contexto entre múltiples procesos UNIX es elevado.
- El planificador del sistema puede manejar eficientemente hasta un número determinado de procesos.
- Las operaciones para usar variables de sincronización compartidas suelen ser *lentas*.
- Además el `fork()` es un mecanismo de creación de procesos *no-estructurado*, que propicia errores en la programación debido a que, a veces, no se sabe bien qué encarnación del proceso original se está ejecutando[10].

Una posible solución sería renunciar a la *protección* derivada de tener espacios de direcciones separados, que se presupone si se programa directamente con los procesos de UNIX. Esto nos lleva al concepto de *hebra* de control independiente dentro de un proceso como entidad de planificación. Los sistemas operativos conformes con POSIX 1003.1c –como Linux– ahora planifican procesos y hebras. Un proceso puede contener ahora una o varias hebras. Por tanto, podemos entender a una hebra como un flujo de control que comparte con otras hebras el texto del proceso al que pertenecen. Cada hebra tiene su propia pila –vacía al inicio–, donde almacena sus variables locales, y comparte con otras hebras la zona de datos –donde se almacenan las variables globales– y el *heap* del proceso al que pertenecen.

[10]para disringuirlos la función `fork()` devuelve el valor `0` si el código que se ejecuta es el del proceso padre

atributos	valor
contentionscope	PTHREAD_SCOPE_PROCESS
detachstate	PTHREAD_CREATE_JOINABLE
stackaddr	NULL
stacksize	NULL
policy	SCHED_OTHER
inheritsched	PTHREAD_EXPLICIT_SCHED

Tabla 1.1: Valores de los campos que componen la estructura atributos

Función de creación de hebras

Cada proceso tiene asociado un texto o *programa* que al principio existe como una sola hebra que ejecuta la función `main()` (en C/C++). Conforme avanza la ejecución del programa anterior, la hebra que inicialmente ejecuta `main()` podría crear 1 ó más hebras asociadas a su mismo proceso. Una hebra que crea a otra designa una función `f()` del texto del proceso para que la ejecute, siguiendo después con su propia ejecución. La hebra creada ejecutará la función `f()` concurrentemente con el resto de las hebras del programa hasta que acabe, o bien se den alguna de las condiciones de terminación previstas.

La función que crea y activa una nueva hebra, utilizando la interfaz POSIX 1003.1c, es la siguiente:

```
int pthr_create(pthread_t *nueva_hebra,
                const pthread_attr_ t *atr,
                void *(*nombre_funcion)(*void),
                void *args )
```

- `pthread_t` es un tipo *opaco* que actúa como un *manejador* ("handle", en inglés) de la hebra, es decir, cuando haya que llamar a las operaciones de la hebra creada se utilizará su manejador como una referencia.

- El atributo de creación de una hebra `atr` se asigna con funciones específicas que modifican una estructura donde se almacenan las características iniciales de creación. Entre éstas se encuentran: tamaño de la pila, de su planificación (*tiempo real* o *tiempo compartido*), si se la puede esperar con la operación `pthread_join()` o si es una hebra independiente o *hebra del sistema*. Los valores por defecto de los atributos cuando se llama a la función de creación con el parámetro `NULL` pueden verse en la tabla 1.1.

- Causas de terminación de una hebra:
  - cuando la función designada en `pthread_create()` acaba su ejecución –sólo para hebras distintas de la inicial,
  - la hebra llama a `pthread_exit` explícitamente en su código,
  - resulta terminada por otra hebra con una llamada a `pthread_cancel()`,
  - el proceso asociado a la hebra termina debido a una llamada a `exit()`,
  - la hebra inicial termina de ejecutar `main()` sin haber llamado a `pthread_exit()`.

- `void * (*nombre_funcion) (* void)` es el nombre de la función que ha de ejecutar la hebra,

- `void * args` puntero que se pasa como parámetro de la función de la hebra y que puede ser NULL.

Ejemplo de creación de hebras POSIX

Se trata de un programa que implementa un planificador de avisos, que se podría utilizar como una agenda de funcionalidad muy básica, con una interfaz de usuario muy simple que sólo admite peticiones de aviso en modo texto.

Funcionalidad del programa:

- se aceptarán continuamente, dentro de un bucle, peticiones del usuario;
- en cada una de dichas peticiones se introduce una línea completa de texto, hasta que se detecte un error o el fin de archivo en `stdin`;
- en cada línea, el primer símbolo que se puede formar es interpretado como el número de segundos que hay que esperar hasta mostrar el aviso;
- el resto de la línea (hasta 64 caracteres) es el propio mensaje de aviso.

```
char linea[128];
paquete_control_t  *pcontrol;
pthread_t hebra;
/* Se abre el fichero que contendr'a las salidas */
if ((fp = fopen("salidas", "w")) == NULL)
    fprintf(stderr, "Error en la apertura del fichero de salida\n"),exit(0);
while (1) {
    printf ("Petici'on de aviso> ");
    if (fgets (linea, sizeof (linea), stdin) == NULL) exit (0);
    if (strlen (linea) <= 1) continue;
    /* asignar memoria din'amica al paquete de control */
    if (sscanf(linea, "%d %64[^\n]", &pcontrol->segundos, &pcontrol->mensaje)<2){
           if (pcontrol->segundos==-1) break;
           else{
              fprintf(stderr, "Entrada err'onea\n");
              free(pcontrol);
       }
     }
     else{
        estado= pthread_create(&hebra, NULL, aviso, pcontrol);
        if (estado != 0) fprintf(stderr, "Error al crear la hebra
             aviso\n"),exit(0);
          }
     }
    if (fclose(fp)) fprintf(stderr, "Error al cerrar el fichero\n");
}
```

Todas las hebras comparten el mismo espacio de direcciones, por lo que habrá que crear una estructura (utilizando memoria dinámica) que contenga los valores del plazo de tiempo y del mensaje del aviso asociados a cada nueva petición del usuario. Un puntero a dicha estructura tiene que ser pasado a una hebra como cuarto argumento de `pthread_create()` en el momento de su creación. Además, no hay necesidad de que la hebra principal reúna el control de las hebras terminadas, ya que éstas se pueden crear con la opción `detached`. De esta forma, los recursos de las hebras serán devueltos automáticamente al sistema cuando el programa termine.

La función aviso es el subprograma común a todas las hebras. La hebra se suspende durante el número de segundos especificado en su paquete de control y cuando ésta pasa de nuevo al estado *activo*, imprime la cadena con el mensaje del usuario.

1.3 Mecanismos de sincronización de bajo nivel en memoria compartida

Para poder implementar mecanismos de sincronización en los lenguajes de alto nivel se utilizan instrucciones de muy bajo nivel que aprovechan la *naturaleza síncrona del hardware*. Dichos mecanismos consisten básicamente en hacer a un proceso ininterrumpible o aprovechar el hecho de que el hardware de memoria sólo puede servir una petición al mismo tiempo (ver sección 1.1.1).

1.3.1 Inhibición de interrupciones

Se utiliza una instrucción para prohibir las interrupciones. Cualquier nueva interrupción es pospuesta hasta que el proceso activo ejecute una instrucción que las vuelva a permitir. Si un proceso antes de ejecutar una sección crítica consigue que se prohiban las interrupciones, entonces tiene la seguridad de estar ejecutando la sección aludida de forma totalmente exclusiva. Sin embargo, otros procesos que pueden ser críticos para el buen funcionamiento del sistema dejarían de ejecutarse hasta que se vuelvan a permitir las interrupciones.

La única ventaja de este método es que es muy rápido: sólo se necesita 1 instrucción máquina. No obstante, presenta los siguientes inconvenientes, algunos de ellos inaceptables, en programación de sistemas:

- Una vez prohibidas las interrupciones no pueden tratarse eventos de tiempo real (p.e. dispositivos hardware que necesitan mucho servicio). Las secciones críticas largas hacen muy difícil obtener buen rendimiento en la programación si se utiliza este método.
- Excluye que actividades no conflictivas puedan entrelazar sus instrucciones.
- Si las secciones críticas se ejecutan sin interrupciones, no pueden utilizarse instrucciones que dependan del reloj dentro de ellas.
- Tiene problemas de seguridad si se permitiera la anidación de secciones críticas.

- Se pueden producir bloqueos si se omite programar la instrucción de permitir de nuevo las interrupciones.

1.3.2 Cerrojos

Están basados en una instrucción explícita de sincronización por la memoria, que se denomina *test_and_set*(TST). Dicha función realiza dos operaciones como una única operación atómica:

1. Lee el valor de la *variable de sincronización*.
2. Le asigna un valor a dicha variable.

Después, devuelve el valor leído –en la primera operación– de la variable de sincronización. No se permite, por tanto, la ejecución de una instrucción atómica de ningún otro proceso en medio de la ejecución de las dos operaciones anteriores.

Su uso previene las *condiciones de carrera* en el acceso a datos compartidos, aunque pueden inducir a la *espera ociosa*[11] de las hebras que esperan leer un valor de la variable de sincronización que les resulte favorable para seguir ejecutando el resto de sus instrucciones.

Operaciones de declaración y acceso a la variable de *sincronización*

Para poder utilizar los cerrojos en la sincronización, la operación TST ha de ser una operación perteneciente al repertorio de instrucciones de bajo nivel de la plataforma donde ejecutemos el programa. Sin embargo, con fines didácticos, podríamos escribirla como la función siguiente:

```
atomic function TST(var c: boolean): boolean;
  begin
    TST:= c;
    c:= TRUE;
  end;
```

Las variables de tipo cerrojo admiten sólo 2 valores. Por consiguiente, en la simulación que estamos haciendo con una función de alto nivel, es práctico declararlas de tipo *boolean*.

```
type variable_sincronizacion= boolean; \\o "cerrojo"

var c: variable_sincronizacion;
...
c:= FALSE;
```

La utilización de los cerrojos para resolver un problema de exclusión mutua se llevaría a cabo de la siguiente manera:

```
\\Protocolo de adquisicion
procedure ComienzaSeccion(var c: cerrojo);
begin
```

[11] se dice que un proceso realiza espera *ociosa* u *ocupada* cuando consume ciclos realizando iteraciones hasta que la condición de dicho bucle cambie de valor

```
  while TST(c) do
      ;
end;
```

<<Sección Crítica a proteger>>

```
\\Protocolo de restitucion
procedure FinalizaSeccion (var c:cerrojo);
begin
   c:= FALSE;
end;
```

Dado que cada sección crítica contiene un conjunto de variables compartidas que es necesario proteger, se asociará una variable de sincronización independiente a cada uno de estos grupos. De esta manera, conjuntos de procesos que adquieran cerrojos diferentes pueden entrelazar sus instrucciones sin tener que esperarse.

Cerrojos POSIX 1003

Si se usa la interfaz POSIX, a los cerrojos se les suele asignar memoria como:

- Estructuras de datos estáticas (o automáticas): `pthread_mutex_t cerrojo;`
- Estructuras de datos a las que se puede asignar memoria dinámicamente:

```
pthread_mutex_t *mp;
...
mp= (pthread_mutex_t *) malloc(sizeof(pthread_mutex_t));
```

Pueden ser inicializados: `pthread_mutex_t cerrojo= PTHREAD_MUTEX_INITIALIZER;`

También dinámicamente: `pthread_mutex_init( &cerrojo, &atr);`

El valor `NULL` inicializa el cerrojo con los valores por defecto de sus atributos. La inicialización de un cerrojo ha de ser única para cada ejecución del programa.

Las operaciones más importantes para programar con cerrojos:

- Bloqueo y desbloqueo de cerrojos: `pthread_ mutex_lock(&cerrojo);`, `pthread_mutex_unlock(& cerrojo);`. Se pueden producir condiciones de error tras la ejecución de la operación de desbloqueo. Esto se controla porque las funciones anteriores devuelven un valor entero, que si es $\neq$ `0` indicará un código de error. Con las operaciones anteriores los procesos que esperan adquirir un cerrojo no realizan espera ociosa. Las hebras de un programa esperan en una cola FIFO hasta que el cerrojo que quieren adquirir se desocupe.
- La función `pthread_mutex_trylock(&cerrojo);` adquiere el cerrojo si está disponible o devuelve `EBUSY`. Hay que programarlo como la condición de un bucle de espera ociosa.
- Destrucción de variables cerrojo con la operación `pthread_mutex_destroy(&cerrojo);`.

Propiedades de los programas que utilizan este mecanismo

Los cerrojos son un mecanismo fácil de implementar y permite verificar fácilmente los programas escritos con él. Es un mecanismo de sincronización que se puede utilizar en mono y multiprocesadores. A diferencia del mecanismo de *inhibir interrupciones*, las operaciones con cerrojos no impiden el acceso simultáneo de los procesos a secciones críticas no relacionadas. Es decir, utilizando esta primitiva de sincronización en el caso de monoprocesadores, los procesos que acceden a dichas secciones críticas pueden entrelazar libremente sus instrucciones, salvo que se programen de forma explícita más sincronizaciones en el código de estos. Si se tienen varias secciones críticas en un programa, se declarará un cerrojo por cada una, para así favorecer el obtener el número máximo de entrelazamientos de instrucciones y, por tanto, optimizar la concurrencia en la ejecución de los procesos.

El inconveniente más importante de los cerrojos consiste en que puede producir la marginación de algunos de los procesos concurrentes del programa, los cuales no consiguen ejecutar instrucciones útiles porque nunca consiguen leer un valor de la variable de sincronización que les sea favorable. Puesto que si hay varios procesos esperando que la variable de sincronización del cerrojo cambie de valor, no se puede saber cuál de ellos consigue el cerrojo primero. Además, si se necesitan varios cerrojos en un programa hay que adquirirlos en orden jerárquico, si no se pueden producir interbloqueos:

```
 P1                        P2
----                      -----
ComienzaSeccion(L1)       ComienzaSeccion(L2)
ComienzaSeccion(L2)       ComienzaSeccion(L1)   Interbloqueo!!!!
    S.C.                      S.C.
      ...                       ...
```

Ejemplo de utilización de cerrojos POSIX

Presentamos ahora una mejora del programa anterior (ver subsección 1.2.2), el cual creaba una una hebra "aviso" para cada petición del usuario. En esta versión se usa una única hebra servidora, que extrae el primer elemento de la lista de peticiones. El programa principal (`main()`), no mostrado, insertaría nuevas peticiones en la citada lista, en el orden de menor tiempo de aviso. La lista ha de estar protegida por un cerrojo (`lista_mutex`), y la hebra servidora se suspende al menos durante un segundo en cada iteración, para asegurar que la hebra `main()` tenga oportunidad de bloquear el cerrojo e insertar una nueva petición a la lista.

Si la lista de peticiones no está vacía, la hebra servidora extrae su primer elemento y determina el tiempo que falta para que se cumpla el aviso. Si el tiempo del aviso ha pasado, entonces le asigna a `sleep_time` el valor 0; si no, calcula el número de segundos que tiene que esperar y se lo asigna igualmente. La llamada a `sched_yield()` proporciona una oportunidad a la hebra `main` para ejecutarse si tiene una entrada pendiente del usuario. Si, por el contrario, el tiempo del aviso no hubiera llegado aún (`sleep_time` $>$ `0`), entonces la hebra servidora se suspende durante el tiempo que falta. Por último, nótese que la hebra servidora desbloquea el cerrojo `lista_mutex`, antes de suspenderse, para que así la hebra `main()` pueda bloquearlo e insertar una nueva petición del usuario.

```
pthread_mutex_t lista_mutex = PTHREAD_MUTEX_INITIALIZER;
paquete_control_t *lista_pet = NULL;

 /*
  * funcion que ejecuta la hebra servidora de avisos.
  */
void *servidora (void *arg){
   ...
   paquete_control_t *pcontrol;

   int sleep_time;
   int estado;
   time_t ahora;
   while (1) {

       estado = pthread_mutex_lock(&lista_mutex);

       if (estado !=0) fprintf(stderr, "Error al bloquear lista_mutex\n"),
                        exit(0);

       /* asignar el puntero a la direccion de comienzo de "lista_pet"*/
       pcontrol = (paquete_control_t*)lista_pet;
       if (pcontrol == NULL)
           sleep_time = 1;
       else{ /* lista con peticiones */
           lista_pet = pcontrol->enlace;
           ahora = time(NULL);
           if (pcontrol->tiempo <= ahora) sleep_time = 0;
           else sleep_time = pcontrol->tiempo - ahora;
       }

       estado = pthread_mutex_unlock(&lista_mutex);

       if (estado != 0) fprintf(stderr, "error al desbloquear mutex"),
                       exit(0);

       if (sleep_time > 0) sleep(sleep_time);
       else sched_yield();

       if (pcontrol != NULL){
           fprintf(fp, "(%d) %s  {%ld}\n", pcontrol->segundos,
                   pcontrol->mensaje, pcontrol->tiempo);
           free(pcontrol);
       }
   }
}
```

1.3.3 Semáforos

Es un tipo abstracto de datos, propuesto por Dijkstra en 1968 [Dijkstra, 1965], para desarrollar sistemas operativos multiusuario de una manera estructurada. Desde entonces se ha venido utilizando como una primitiva de sincronización de procesos bastante potente para programación de sistemas.

Los semáforos son una primitiva pensada para sincronizar a los procesos a través de memoria común. Si en un programa se necesitan varios semáforos, se declararán como variables de ese tipo. Se pueden utilizar tanto para definir secciones críticas en los programas como para sincronizar procesos. Además, introducen sólo la sincronización necesaria entre procesos concurrentes, ya que procesos que utilicen semáforos diferentes se pueden ejecutar independientemente y sin ninguna restricción de sincronización.

Definición y operaciones sobre un semáforo

Un semáforo es un *tipo de dato abstracto* (TDA) que sólo tiene definidos valores no negativos y para el que se definen 3 tipos de operaciones:

- Inicialización: se ejecuta 1 sola vez al principio del programa; los valores de inicialización han de ser no negativos.
- *wait(s) ó P(s)*:

  ```
  si s > 0 entonces s := s - 1
  si no, bloquear al proceso en una cola
  ```

- *signal(s) ó V(s)*:

  ```
  comprobar si hay procesos bloqueados,
  si los hay, entonces desbloquear uno (no necesariamente el que lleve mas
    tiempo bloqueado)
  si no s := s + 1
  ```

No hay más operaciones definidas sobre los semáforos, p.e., no es legal comprobar el valor de una variable semáforo. Los semáforos suelen ser implementados en el núcleo del sistema operativo y sus operaciones vienen dadas como otras llamadas al sistema. Se debe evitar el ejecutar operaciones de un semáforo innecesariamnte, por ejemplo, ejecutar `signal(s)` si la cola de `s` está vacía.

En el ejemplo siguiente se pretende calcular la suma de los primeros múltiplos de 5. Para lo cual se utilizan 2 procesos: el primero calcula la sucesión de los naturales, el segundo escribe los múltiplos de 5 y halla la suma de los múltiplos encontrados hasta ese momento. Se detiene al llegar al décimo múltiplo de 5.

```
var S1, S2, N: semaforo;
    s, suma: integer;
    s:=0; suma:=0;
```

```
inic(S1, 1); inic(S2, 0); inic(N,10);

 P1                    P2
----                  ----
while TRUE do         var s0: integer;
 begin                while TRUE do begin
  wait(S1);            wait(N);
   s:= s+1;             wait(S2);
   if (s mod 5=0)        suma:= suma + s;
    then signal(S2)      s0:= s;
    else signal(S1)     signal(S1);
end;                    Escribir(s0);
                       end;
```

Atomicidad de las operaciones

Las operaciones sobre la variable de sincronización se ejecutan atómicamente, porque si se permitiera el entrelazamiento de las instrucciones de varias operaciones sobre el mismo semáforo, el valor final de la variable sería impredecible y no se cumplirían ni las propiedades de seguridad, ni las de vivacidad del programa que los utilizase.

Tipos de semáforos

Los semáforos se clasifican según los valores que pueda tomar la variable de sincronización y según para lo que se utilicen.

Atendiendo al rango de valores:

1. *Semáforos binarios:* sólo pueden tomar los valores 0 y 1.
2. *Semáforos generales:* pueden tomar valores no negativos.

Atendiendo a su utilización:

1. *Exclusión mutua:* Se inicializan a 1 y se bloquea el segundo proceso que intente ejecutar la operación wait (si ahora el valor es 0).
2. *Sincronización:* Se inicializan a 0. Se bloquea el primer proceso que intente ejecutar wait.

Propiedades de los programas que utilizan esta primitiva

Los semáforos evitan que los procesos realicen espera ociosa. Cuando varios procesos esperan que se dé una determinada condición se les bloquea, tras ejecutar la instrucción `wait()`, en la cola del semáforo. Por tanto, se pueden ejecutar simultáneamente operaciones sobre distintos semáforos. Además, si se declaran varios semáforos en un programa, estos han de utilizarse jerárquicamente; si no, se pueden producir bloqueos.

El mayor inconveniente se debe a que son unas primitivas difíciles de utilizar correctamente. Además, pueden producir la inanición de los procesos, ya que la operación signal no asegura qué proceso será desbloqueado primero.

```
var S1, S2: semaforo;       |   var S, S1: semaforo;
Inic(S1, 1), Inic(S2, 1);   |   Inic(S1, 0); Inic(S, 1);
    P1            P2        |        P1          P2
  -----         -----       |    ---------    ---------
  wait(S1);      wait(S2);  |     wait(S);     wait(S);
  wait(S2);      wait(S1);  |                  wait(S1);
       interbloqueo!!!      |          interbloqueo!!!
    .              .        |
    .              .
    .              .
```

Semáforos POSIX 1003

Los semáforos son una facilidad de sincronización opcional en la interfaz POSIX para programar con hebras. No todas las implementaciones de la interfaz de las hebras soportarán semáforos. Para comprobarlo, sólo hay que comprobar si la macro `_POSIX_SEMAPHORES` está definida en el archivo `<unistd.h>`.

Los semáforos pertenecen al antiguo estándar `P1003.b`, en lugar del nuevo estándar `P1003.1c`, por lo tanto, su interfaz tiene un estilo ligeramente diferente al de las otras variables de sincronización POSIX.

Tipos

Los semáforos en POSIX pueden ser de 2 tipos:

- *no-nombrados*: similares a las otras variables de sincronización de las hebras.
- *nombrados*: se les asocia una cadena globalmente conocida en el sistema (similar a un `pathname`),

A los semáforos *no-nombrados* se les asigna memoria de un proceso y se les inicializa. Pueden ser utilizados por más de un proceso, aunque esto depende de cómo sean inicializados y se les asigne memoria.

Los semáforos *nombrados* son siempre compartidos por varios procesos. Tienen un identificador de usuario, identificador de grupo y protección igual que los ficheros regulares del sistema. La implementación de estos semáforos puede asociar a cada semáforo un fichero real, o bien simplemente utilizar la cadena asociada al semáforo como un nombre interno dentro del núcleo del SO. Para que pueda ser transportable, el nombre de un semáforo ha de comenzar con el carácter _ y no debe incluir otros caracteres _ dentro de dicho nombre.

El espacio de nombres es compartido por todos los procesos del sistema, de tal forma que si un conjunto de procesos utiliza el mismo nombre, entonces obtiene el mismo semáforo.

Ambos tipos de semáforos son representados mediante el tipo `sem_t` y todas las rutinas de los semáforos están definidas en el fichero `<semaphore.h>`.

Operaciones de inicialización definidas sobre semáforos no-nombrados

La inicialización de los semáforos *no-nombrados* se realiza mediante la operación:

`int sem_init(sem_t * semaforo,` *int pcompart*, *unsigned int contad*`)`

El valor inicial del semáforo se le asigna al argumento *contad* de la llamada a la función anterior. Si argumento *pcompart* es distinto de cero, entonces el semáforo puede ser utilizado por hebras que residen en procesos diferentes; si no, sólo puede ser utilizado por hebras dentro del espacio de direcciones de un único proceso.

Un semáforo no-nombrado puede ser destruido mediante la operación:

`int sem_destroy(sem_t *semaforo)`

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. Esta operación devuelve un error si la implementación detecta que el semáforo está siendo utilizado por otras hebras bloqueadas, tras haber llamado a la operación `sem_wait`.

Operaciones de inicialización definidas sobre semáforos nombrados

Lo primero que hay que hacer es establecer una conexión entre el semáforo y el proceso llamador para poder realizar sobre el semáforo operaciones adicionales. El semáforo permanece utilizable hasta que es cerrado.

`sem_t *sem_open(const char*`*nombre*`, int` *oflag*
[, `unsigned long` *modo*, `unsigned int` *valor*])

La operación anterior devuelve la dirección del semáforo al proceso llamador. *nombre* apunta a una cadena que nombra al objeto semáforo. Si hay situación de error, devuelve `-1` y asigna `errno` para indicar la condición de error.

Si un proceso hace varias llamadas a `sem_open` con el mismo valor de *nombre*, siempre se le devuelve la misma dirección del semáforo.

La bandera *oflag* determina si el semáforo es creado o accedido mediante la llamada `sem_open`. Los valores válidos de `oflag` son: `0`, `O_CREAT` (crea un semáforo, si no existe ya), `O_CREAT | O_EXCL` (falla si el semáforo ya existe).

Después de haberse creado un semáforo con el flag `O_CREAT` y haberle dado un nombre, los otros procesos pueden conectar con dicho semáforo llamando a `sem_open()`, utilizando el valor `nombre` y sin asignar bits en `oflag`.

Si se utiliza la bandera `oflag`, entonces hay que asignar 2 argumentos más:

- `modo` (*3er argumento*): Fija los permisos del semáforo, modificándolos tras borrar todos los bits asignados en la máscara de creación del fichero del proceso.
- `valor` (*4o argumento*): se crea el semáforo con un valor inicial. `valor` debe ser menor o igual que `SEM_VALUE_MAX`

La operación: `int sem_close(sem_t *semaforo)` destruye la conexión con un semáforo nombrado. Esta operación no debe ser utilizada en semáforos no-nombrados.

Operaciones de sincronización

Para cualquier tipo de semáforos las hebras para sincronizarse con una condición llaman a la

función siguiente pasándole un identificador de semáforo inicializado con el valor `0`:

```
int sem_wait(sem_t *semaforo)
```

Si el valor de *semaforo* fuera distinto de `0`, entonces como resultado de la ejecución de esta función el valor de `s` se decrementa en una unidad y no bloquea a la hebra que llama.

La operación contraria: `int sema_post(sem_t *semaforo)` sirve para señalar a las hebras bloqueadas en un semáforo. Si no hay hebras bloqueadas en este semáforo, entonces simplemente incrementa el valor del semáforo. No existe ningún orden de desbloqueo definido si hay varias hebras esperando en un semáforo, es decir, la implementación de la operación anterior puede escoger para desbloquear a cualquiera de las hebras que esperan. En particular, otra hebra ejecutándose puede decrementar el valor del semáforo antes de que cualquier hebra despertada lo pueda hacer, posteriormente se volvería a bloquear la hebra despertada. Las dos funciones anteriores pueden devolver errores si el semáforo ha sido inicializado incorrectamente.

La implementación interna de las operaciones anteriores es *asíncrona segura*, esto quiere decir que si una señal interrumpe a una operación mientras posee el acceso exclusivo a la variable semáforo, dicha variable no queda inaccesible para operaciones del semáforo posteriores, es decir, existe atomicidad de las operaciones frente a señales asíncronas exteriores.

Algunas veces es conveniente intentar evitar el bloqueo que produce la llamada a la operación `sem_wait` cuando el valor de la variable protegida del semáforo es `0`, utilizando alternativamente otra función: `intsem_trywait(sem_t *semaforo)`. La función anterior decrementa atómicamente el semáforo sólo si es mayor que `0`, si no devuelve un error.

La siguiente función: `int valorp = sem_trywait(sem_t *semaf)` sirve para obtener el valor actual del semáforo `semaf`. Tras ejecutarse completamente, almacena dicho valor en la posición apuntada por `valorp`.

Ejemplo con semáforos POSIX
En el siguiente ejemplo se presentan 2 hebras, una que escribe en la variable global `dato_protegido` y la otra que lee dicha variable. Cada hebra realiza 10000 iteraciones de un bucle en el que las hebras escriben o leen repetidamente el valor de dicha variable compartida. Se declaran e inicializan 2 semáforos *no–nombrados*: `escribir_ok` y `leer_ok` para escribir o leer, respectivamente, el valor de la variable. Inicialmente sólo puede actuar la hebra escritora y por eso se pasa el valor 1 en el tercer argumento de `sem_init()`. De manera análoga, inicialmente no se puede leer la variable compartida porque todavía no se ha escrito nada en ella. Por tanto, se pasa el valor 0 en el tercer argumento de la función `sem_init()` para los semáforos `leer_ok` y `mutex`.

El objetivo del semáforo `mutex` es asegurar que no se producen errores en las salidas por pantalla, ya que en el acceso a dicho recurso se pueden producir *condiciones de carrera* entre las hebras del programa.

```
sem_t escribir_ok, // inicializado a 1
      leer_ok, // inicializado a 0
      mutex ; // inicializado a 1
unsigned long dato_protegido ; // valor para escribir o leer
const unsigned long num_iter = 10000 ; // nmero de iteraciones

void* escribir( void* p ){
  unsigned long contador = 0 ;
  for( unsigned long i = 0 ; i < num_iter ; i++ ){
    contador = contador + 1 ; // genera un nuevo valor
    sem_wait( &escribir_ok ) ;
    dato_protegido = contador ; // escribe el valor
    sem_post( &leer_ok ) ;
    sem_wait( &mutex ) ;
    cout << "dato escrito == " << contador << endl << flush ;
    sem_post( &mutex ) ;
  }
  return NULL ;
}

void* leer( void* p ){
  unsigned long valor_leido ;
  for( unsigned long i = 0 ; i < num_iter ; i++ ){
    sem_wait( &leer_ok ) ;
    dato_leido = dato_protegido ; // lee el valor generado
    sem_post( &escribir_ok) ;
    sem_wait( &mutex ) ;
    cout << " valor leido == " << valor_leido << endl << flush ;
    sem_post( &mutex ) ;
  }
  return NULL ;
}

int main(){
  pthread_t hebra_escritora, hebra_lectora ;
  sem_init( &mutex, 0, 1 ) ;
  sem_init( &escribir_ok, 0, 1);
  sem_init( &leer_ok, 0, 0);
  pthread_create( &hebra_escritora, NULL, escribir, NULL );
  pthread_create( &hebra_lectora, NULL, leer, NULL );
  pthread_join( hebra_escritora, NULL ) ;
  pthread_join( hebra_lectora, NULL ) ;
  sem_destroy( &escribir_ok);
  sem_destroy( &leer_ok );
}
```

1.4 Propiedades de los sistemas concurrentes

En los sistemas [12] concurrentes una *propiedad* se entiende como un atributo que se cumple en toda ejecución perteneciente al *comportamiento* del sistema.

Cualquier propiedad de un sistema concurrente –por compleja que sea– podría ser formulada como una combinación de 2 tipos de propiedades fundamentales:

- *Seguridad*[13]: una propiedad de este tipo afirma que ninguna ejecución incluida en el comportamiento del sistema puede llegar a entrar en un estado prohibido (o *no deseado*), por ejemplo, que el sistema nunca entre en una situación de interbloqueo de todos sus procesos.

- *Vivacidad*[14]: una propiedad de este tipo afirma que el sistema finalmente entrará en un estado *deseado*, por ejemplo, alcanzar la sección crítica por parte de un proceso.

1.4.1 Propiedades de seguridad

Las propiedades de seguridad expresan determinadas condiciones[15] que se han de cumplir durante toda la ejecución del sistema. Suelen expresar condiciones de exclusión mutua, relaciones de precedencia entre instrucciones o procesos, ausencia de interbloqueos, etc. Una regla práctica para diferenciar las propiedades de seguridad del resto de propiedades consiste en comprobar si implementándolo como un sistema secuencial se podría afirmar que se cumple la propiedad referida. Por ejemplo, la imposibilidad de llegar a una situación de interbloqueo[16] entre los procesos de un sistema concurrente es una propiedad muy importante de seguridad. Se puede comprobar que efectivamente se trata de una propiedad de seguridad porque una versión secuencial del sistema estaría libre de interbloqueo.

Ejemplos de propiedades de seguridad:

1. *Problema de la exclusión mutua*: 2 procesos no pueden ejecutar simultáneamente las instrucciones de una sección crítica.

2. *Problema del productor-consumidor*: el proceso consumidor no puede retirar datos de un bufer vacío, análogamente un proceso productor no puede insertar datos en un buffer lleno.

3. *Interbloqueo*: un conjunto de procesos mantienen sus procesadores al mismo tiempo que intentan escribir datos en memoria, la memoria se llena y no hay ningún procesador disponible para escribir en el disco.

[12] aquí entenderemos un *"sistema"* como un conjunto de componentes activos que interaccionan, mediante comunicación y sincronización. Incluiría a los programas y aplicaciones concurrentes y de tiempo real, pero no sólo; también se consideran partes del sistema dispositivos hardware controlados por el software.

[13] en inglés se utiliza el término "safety" para denominarla

[14] en inglés se la denomina "liveness"

[15] se denominan *especificaciones estáticas* del sistema

[16] *deadlock*

1.4.2 Propiedades de vivacidad

Las propiedades de vivacidad sirven para expresar que el sistema llegará a cumplir determinada condición[17] con el tiempo, aunque no podemos precisar cuánto tardará en conseguirlo.

Ejemplos de propiedades de vivacidad:

- *Problema de la exclusión mutua*: si un proceso desea entrar en una sección crítica, no puede estar esperando siempre, alguna vez conseguirá entrar.
- *Problema del productor-consumidor*: un proceso que quiera introducir o eliminar datos del bufer, lo conseguirá en un tiempo no infinito.

Normalmente el incumplimiento de la propiedad de vivacidad ocasiona la *inanición*[18] de 1 ó más procesos del sistema concurrente. Se dice que un proceso sufre inanición si es indefinidamente pospuesto por los otros y nunca consigue llegar a realizar completamente la tarea para la que fue programado. Esta situación es menos grave que el interbloqueo porque existen procesos del sistema que realizan trabajo útil, pero no sería admisible que ocurriera. Un sistema concurrente sólo será completamente correcto cuando se demuestre que los procesos no sufren inanición en ninguna de sus posibles ejecuciones.

1.4.3 Propiedad de equidad

Además de la propiedad de vivacidad se ha de asegurar que cuando un proceso esté preparado para ejecutarse debe hacerlo con justicia relativa respecto de los demás procesos. Es una propiedad bastante más exigente que la vivacidad. El que pueda ser demostrada o no suele depender de la implementación de los mecanismos de sincronización a bajo nivel que tenga el sistema de ejecución del lenguaje o el planificador de procesos de la plataforma concreta.

1.4.4 Verificación

Un programa secuencial se dice que es *"parcialmente correcto"* si el programa termina con los resultados esperados, siendo la terminación una premisa no exigida. Adicionalmente, se dice que un programa es *totalmente correcto* si es parcialmente correcto y se puede demostrar que siempre termina.

Sin embargo, los conceptos anteriores no son adecuados para definir la corrección de los sistemas concurrentes, ya que la implementación de estos sistemas asume que están en ejecución permanente. De hecho, el fin de una ejecución se suele asociar a una condición de error o a un reinicio forzado. Ejemplos de sistemas que no terminan y que son sistemas intrínsecamente concurrentes: los sistemas operativos, los sistemas de control de tiempo real, cajeros automáticos, control de vuelos y reservas, etc.

[17] también se las denomina como *especificaciones dinámicas* del sistema.
[18] *starvation* en inglés

La definición más general de corrección del software que podemos enunciar, y que puede ser aplicada tanto a sistemas secuenciales como concurrentes, es la siguiente: *un sistema es correcto si satisface sus propiedades previamente especificadas.*

Para llevar a cabo la demostración de que un código es correcto –verificación del software– se pueden emplear diferentes métodos:

- Depuración del código[19]: consiste en explorar algunas de las ejecuciones del código y verificar que son aceptables –se cumplen las propiedades. El problema de este método es que nunca puede demostrar la ausencia de errores transitorios en un software.

- Razonamiento operacional: se podría entender como un "análisis de casos exhaustivo", es decir, se explorarían todas las posibles secuencias de ejecución del código, considerando todos los posibles entrelazamientos de las operaciones atómicas de los procesos. Esto es inviable debido al número astronómico de secuencias de entrelazamiento que incluso un fragmento corto de software puede producir.

- Razonamiento *asertivo*: se trata de un análisis abstracto basado en la Lógica Matemática que permite obtener una representación de los estados concretos que un programa alcanza durante su ejecución. Un estado viene definido por los valores que tienen las variables del programa en él.

Verificación de programas basada en razonamiento asertivo

Se basa en la utilización de un *sistema lógico formal* (SLF) que facilita la elaboración de proposiciones ciertas, con una base lógica precisa, de los estados de un programa. Un estado viene definido por los valores que tienen las variables del programa en él. La definición formal de un SLF es la siguiente:

$$\text{SLF} = \{\text{Símbolos, Fórmulas, Axiomas, Reglas de Inferencia}\}$$

- Símbolos: {sentencias del lenguaje, variables proposicionales, operadores, etc.}

- Fórmulas: secuencias de símbolos *bien formadas.*

- Reglas de Inferencia: indican cómo derivar fórmulas ciertas a partir de axiomas (fórmulas que se sabe son ciertas) y de otras fórmulas que se han demostrado ciertas.

Las reglas de inferencia poseen el siguiente significado: si todas sus hipótesis son ciertas, entonces su conclusión también lo es:

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

Tanto las hipótesis como la conclusión han de ser fórmulas o una representación esquemática de ellas. Un teorema o aserto es una fórmula que representa a una afirmación cierta que pertenece al dominio del discurso. Los asertos del *SLF* que vamos definir coinciden con las líneas o *sentencias lógicas* en las que se estructura la demostración de un programa. Una demostración de corrección de un programa es una secuencia de asertos, tal que cada uno de ellos puede ser derivado de los anteriores mediante la aplicación de una regla de inferencia.

[19] *debugging* o *testing*, en inglés

- Interpretación: para conocer si un aserto dado es cierto es necesario proporcionar una interpretación a las fórmulas del *SLF*, que se define como la siguiente correspondencia:

$$Interpretación \rightarrow \{V, F\}$$

- Seguridad: el SLF que estamos definiendo es seguro respecto a una interpretación si todos los asertos que se pueden derivar con este sistema son hechos ciertos. Si definimos los siguientes conjuntos:

$$hechos = \{certezas\ que\ se\ expresan\ como\ fórmulas\}$$

$$asertos = \{conjunto\ de\ fórmulas\ demostrables\}$$

  Entonces se cumplirá la siguiente relación de inclusión:

$$asertos \subseteq hechos$$

  Sólo si el *SLF* posee la propiedad de seguridad.

- Complección: un SLF posee esta propiedad si todo aserto cierto es demostrable, esto es:

$$hechos \subseteq asertos$$

Lógica Proposicional

Se trata de un claro ejemplo de *SLF* que formaliza lo que normalmente llamamos el razonamiento basado en el *sentido común*. Las fórmulas de esta lógica se llaman *proposiciones* y sus símbolos son:

- constantes proposicionales $\{V, F\}$,
- las variables proposicionales: $\{p, q, r, \ldots\}$,
- los operadores: $\{\neg, \wedge, \vee, \rightarrow \ldots\}$,
- expresiones que utilizan constantes, variables y operadores.

Interpretación de una fórmula proposicional

Dado un estado s de un programa, descrito por una fórmula P, en el que reemplazamos cada variable proposicional p por su valor en dicho estado y luego utilizamos la tabla de verdad de los conectores lógicos para obtener el resultado. La certeza de la mencionada fórmula P dependerá del estado, de acuerdo con las siguientes definiciones:

- una fórmula se *satisface* en un estado "s" sii posee una interpretación cierta en dicho estado,
- *fórmula satisfascible* sii existe algún estado de programa en el cual la fórmula se puede satisfacer,
- *fórmula válida* sii se puede satisfacer en cualquier estado.

A las proposiciones válidas se les llama tautologías. En una lógica proposicional que cumple con la propiedad de seguridad, todos los axiomas son tautologías, ya que estos han de ser siempre válidos.

Tautologías o leyes de equivalencia proposicionales más utilizadas
Se trata de leyes de equivalencia que permiten reemplazar una proposición por su equivalente y, de esta forma, permiten la simplificación de fórmulas complejas:

1. Ley de negación: $\mathrm{P} = \neg(\neg\mathrm{P})$
2. Ley de los *medios excluidos*: $\mathrm{P} \vee \neg\mathrm{P} = V$
3. Ley de contradicción: $\mathrm{P} \wedge \neg\mathrm{P} = F$
4. Ley de implicación $\mathrm{P} \rightarrow \mathrm{Q} \equiv \neg\mathrm{P} \vee \mathrm{Q}$
5. Ley de igualdad $(\mathrm{P} \rightarrow \mathrm{Q}) \wedge (\mathrm{Q} \rightarrow \mathrm{P}) \equiv (\mathrm{P} = \mathrm{Q})$
6. Leyes de simplificación *Or*:
   - $\mathrm{P} \vee \mathrm{P} = \mathrm{P}$
   - $\mathrm{P} \vee V = V$
   - $\mathrm{P} \vee F = P$
   - $\mathrm{P} \vee (\mathrm{P} \wedge \mathrm{Q}) = \mathrm{P}$
7. Leyes de simplificación-*And*:
   - $\mathrm{P} \wedge \mathrm{P} = \mathrm{P}$
   - $\mathrm{P} \wedge V = P$
   - $\mathrm{P} \wedge F = F$
   - $\mathrm{P} \wedge (\mathrm{P} \vee \mathrm{Q}) = \mathrm{P}$
8. Leyes conmutativas:
   - $(\mathrm{P} \wedge \mathrm{Q}) = (\mathrm{Q} \wedge \mathrm{P})$
   - $(\mathrm{P} \vee \mathrm{Q}) = (\mathrm{Q} \vee \mathrm{P})$
   - $(\mathrm{P} = \mathrm{Q}) = (\mathrm{Q} = \mathrm{P})$
9. Leyes asociativas:
   - $\mathrm{P} \wedge (\mathrm{Q} \wedge \mathrm{R}) = (\mathrm{P} \wedge \mathrm{Q}) \wedge \mathrm{R}$
   - $\mathrm{P} \vee (\mathrm{Q} \vee \mathrm{R}) = (\mathrm{P} \vee \mathrm{Q}) \vee \mathrm{R}$
10. Leyes distributivas:
    - $\mathrm{P} \vee (\mathrm{Q} \wedge \mathrm{R}) = (\mathrm{P} \vee \mathrm{Q}) \wedge (\mathrm{P} \vee \mathrm{R})$
    - $\mathrm{P} \wedge (\mathrm{Q} \vee \mathrm{R}) = (\mathrm{P} \wedge \mathrm{Q}) \vee (\mathrm{P} \wedge \mathrm{R})$

11. Leyes de Morgan:
    - $\neg(P \wedge Q) = \neg P \vee \neg Q$
    - $\neg(P \vee Q) = \neg P \wedge \neg Q$
12. Eliminación-*And*: $(P \wedge Q) \rightarrow P$
13. Eliminación-*Or*: $P \rightarrow (P \vee Q)$

La regla 12 se puede explicar diciendo que el conjunto de estados que cumplen P incluye al conjunto de los que cumplen $P \wedge Q$; se dice que la proposición P es más débil y, por tanto, se *ve implicada* por la expresión (*más fuerte*) $P \wedge Q$. De acuerdo con lo anterior, la constante F es la proposición *más fuerte*, ya que implica a cualquier otra de la lógica. La constante V es la proposición más débil, ya que sería *implicada* por cualquier proposición.

Lógica de Programas

SLF que permite hacer afirmaciones precisas acerca de la ejecución de un programa. Los símbolos de la Lógica de Programas (LP) incluyen a las sentencias de los lenguajes de programación y sus fórmulas, que se denominan *triples*, tienen la forma $\{P\}\,S\,\{Q\}$; donde P y Q son asertos, S es una sentencia simple o estructurada de un lenguaje de programación. Las variables libres de P y Q pertenecen al programa o son *variables lógicas*. Esta últimas actúan como un recipiente de los valores de las variables *comunes* del programa y no pueden ser asignadas más de una vez, esto es, mantienen siempre el valor al que fueron asignadas la primera vez. Aparecen sólo en asertos, no en las sentencias del lenguaje de programación y se suelen representar con letras mayúsculas para distinguirlas de la variables del programa.

Interpretación de los triples

$\{P\}\,S\,\{Q\}$ se interpreta diciendo que será cierto siempre que la ejecución de S comience en un estado del programa que satisfaga el aserto P (*precondición*) y que el estado final, después de cualquier entrelazamiento de instrucciones atómicas resultado de ejecutar S, ha de satisfacer el aserto Q (*poscondición*). Un aserto caracteriza un estado *aceptable* del sistema, es decir, un estado que podría ser alcanzado por el programa si sus variables tomaran unos determinados valores. Cada estado del programa ha de satisfacer su aserto asociado para que la interpretación del triple proporcione el valor V (*verdadero*). Por lo tanto, el aserto representado por la constante lógica V caracteriza a todos los estados del programa, ya que dicho aserto se cumple en cualquier estado del programa independientemente de los valores que tomen las variables. Por otra parte, un aserto que sea equivalente a la constante lógica F no se cumple en ningún estado del programa.

Axiomas y reglas de inferencia de la Lógica de Programas

1. Axioma de la sentencia nula $\{P\}\,\text{null}\,\{P\}$: si el aserto es cierto antes de ejecutarse la sentencia nula, permanecerá como cierto cuando dicha sentencia termine.

   Sustitución textual: $\{P_e^x\}$ es el resultado de sustituir la expresión e en cualquier aparición libre de la variable x en P. Los nombres de las variables libres de la expresión e no deben entrar en conflicto con las variables ligadas que existan en P. Su significado es que cualquier relación del estado de programa que tenga que ver con la variable x y que sea cierto después de la asignación, también ha de haber sido cierto antes de la asignación.

2. Axioma de asignación $\{\mathrm{P}_{\mathrm{e}}^{\mathrm{x}}\}\,\mathrm{x} := \{\mathrm{P}\}$: una sentencia de asignación asigna un valor e a una variable x y, por lo tanto, generalmente, cambia el estado del programa. Una asignación cambia sólo el valor de la variable objetivo, el resto de las variables conservan los mismos valores que antes de la asignación. $\{\mathrm{V}\}\,\mathrm{x} := 5\ \{\mathrm{x} = 5\}$ es un *aserto* (triple cierto), ya que $\{\mathrm{x} = 5\}_5^{\mathrm{x}} \equiv V$.

   Existe una regla de inferencia, en la Lógica de Programas, para cada una de las sentencias que afectan al flujo de control en un programa secuencial estructurado. Así como una regla de inferencia adicional para *conectar* los triples en las demostraciones de los programas.

3. Regla de la consecuencia (1)

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

   el significado de esta regla es que siempre se puede hacer más débil la postcondición de un triple y que su interpretación se mantenga (que el triple siga siendo cierto).

4. Regla de la consecuencia (2)

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

   el significado de esta regla es que siempre se puede hacer más fuerte la precondición de un triple y que su interpretación se mantenga (que el triple siga siendo cierto).

5. Regla de la composición

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

   permite obtener la poscondición y la precondición de 2 sentencias juntas, a partir de la precondición de la primera y de la poscondición de la segunda, si la poscondición de la primera coincide con la precondición de la segunda.

6. Regla del *if*

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

   suponemos que la precondición de la sentencia (if) que queremos demostrar es $\{P\}$ y la poscondición a la que queremos llegar es $\{Q\}$, entonces, para demostrarlo, sólo debemos probar que las dos ramas del if hacen cierta la misma postcondición $\{Q\}$.

7. Regla de la iteración

$$\frac{\{I \wedge B\}\,S\,\{I\}}{\{I\}\ \mathit{while}\ B\ \mathit{do}\ S\ \mathit{enddo}\,\{I \wedge \neg B\}}$$

   Una sentencia {while} podrá iterar un número arbitrario de veces, incluso 0. Por esa razón la regla de la inferencia iterativa está basada en un invariante del bucle: un aserto I que se satisface antes y después de cada iteración del bucle.

Verificación de sentencias concurrentes y de sincronización

En la verificación de los programas concurrentes se produce el problema conocido como de la *interferencia*, que cuando ocurre invalida las demostraciones individuales de los procesos. Se debe a que un proceso puede ejecutar una instrucción atómica que haga falsa la precondición (o poscondición) de una sentencia de otro proceso mientras está siendo ejecutada concurrentemente con el primero. Si esto se diera, el sistema LP dejaría de cumplir la propiedad de seguridad y no nos serviría para verificar programas concurrentes.

El siguiente programa es un ejemplo de interferencia entre procesos de un programa concurrente. La ejecución del mismo puede producir como resultado final $x \in \{0, 1, 2, 3\}$, dependiendo que se cargue (`load y`) y se incremente (`add z`) el registro [20] antes, después o al mismo tiempo, que se ejecutan las 2 operaciones de asignación ($y := 1$ y $z := 2$) del componente derecho de la instruccción `||` de composición concurrente de los procesos:

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

Una peculiaridad del programa anterior es que puede producir el valor final de $x = 2$, a pesar de que $z + y = 2$ no se corresponde con ningún estado del programa.

Hemos de tener en cuenta que no todas las secuencias de entrelazamiento de las instrucciones de los procesos resultan ser aceptables. Por consiguiente, si utilizamos bien las sentencias de sincronización en nuestros programas, se puede evitar la mencionada interferencia. En los lenguajes de programación concurrentes, para poder programar correctamente, se suelen utilizar variantes de las siguientes construcciones sintácticas que consiguen evitarla:

- *Secciones críticas* en el código de los procesos, que han de ser ejecutadas respetando la propiedad de exclusión mutua. De esta forma, se combinan *instrucciones atómicas simples* en acciones atómicas compuestas que se ejecutan de forma indivisible.
- Operaciones (atómicas) de *sincronización con una condición* que retrasan la ejecución de un proceso hasta que se satisface un aserto, indicando que el programa ha alcanzado un determinado estado que permite continuar al proceso retrasado sin que se produzca interferencia. Un ejemplo de esto serían las operaciones de los semáforos `sem_post()` y `sem_wait()`, o las señales de los monitores.

Acción atómica elemental
Se trata de una instrucción abstracta $< \ldots >$, que admite diferentes realizaciones en los lenguajes concurrentes (*bloques sincronizados*, *regiones críticas*, etc.). En los programas secuenciales, las asignaciones siempre son acciones atómicas, puesto que no hay ningún estado intermedio visible al resto de los procesos; sin embargo, esto no ocurre en los programas concurrentes pues, a menudo, una asignación es equivalente a una secuencia de operaciones atómicas elementales.

La declaración de una acción atómica elemental en el texto de un proceso tiene como resultado que cualquier estado intermedio que pudiera existir durante la ejecución de dicha sentencia no sería visible para el resto de los procesos del programa. Por consiguiente, lo

[20] $(x := y + z \equiv \text{load}\, y;\ \text{add}\, z;\ \text{store}\, x)$

importante es que una acción de este tipo realiza una transformación indivisible del estado del programa. De ahí que el proceso P_1 del ejemplo sólo "vea" 2 estados posibles antes y después de la acción $< x := x + 2 >$, es decir, la precondición $\{x = 0\}$ y la postcondición $\{x = 2\}$ de la acción atómica incluida en el proceso P_2.

$$\{x = 0\}$$
$$\text{COBEGIN}$$
$$\{x = 0\}\, P_1 ::< x := x + 1; >\, \{x = 1\} \| \{x = 0\}\, P_2 ::< x := x + 2 >\, \{x = 2\}$$
$$\text{COEND}$$
$$\{x = 3\}$$

Algo totalmente equivalente le ocurre al proceso P_2 respecto de P_1. Como consecuencia de ello la precondición y poscondición de ambos procesos se convierten en disyunciones y se pueden aplicar las reglas de inferencia.

Regla de inferencia de la composición concurrente

De una manera más formal, la acción de asignación a no interfiere con el aserto crítico C si el triple: $\{C \wedge pre(a)\}\, a\, \{C\}$ puede demostrarse como cierto con las reglas y axiomas de la LP. El significado de la afirmación anterior sería que la certeza del aserto C es invariante con respecto a la ejecución de la acción atómica elemental a por otro proceso. La cual, para poder ejecutarse, ha de iniciarse en un estado que satisfaga su precondición ($pre(a)$). Si fuera necesario, para poder llevar a cabo la demostración correctamente, habría que renombrar las variables locales de C para evitar la colisión entre las variables locales de a y de $pre(a)$.

Se dice que un conjunto de procesos está libre de interferencia si no existe ninguna acción elemental dentro de ningún proceso que interfiera con algún aserto crítico de otro proceso. Esta afirmación constituye el antecedente de la siguiente regla de inferencia:

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

El significado de la regla anterior es que si un conjunto de procesos concurrentes está libre de interferencia, entonces su composición concurrente transforma la conjunción de las precondiciones de los procesos en la conjunción de sus poscondiciones Es decir, las demostraciones de los procesos individuales, como programas secuenciales, son válidas, incluso si dichos procesos se ejecutan concurrentemente y no es necesario realizar ninguna demostración adicional. Aplicándolo al ejemplo anterior, como los asertos de los procesos P_1 y P_2 no se invalidan entre sí, la siguiente demostración es válida.

$$\{x = 0\}$$
$$\text{COBEGIN}$$
$$\{x = 0 \vee x = 2\} \qquad \{x = 0 \vee x = 1\}$$
$$x := x + 1; \;\|\; x := x + 2$$
$$\{x = 1 \vee x = 3\} \qquad \{x = 2 \vee x = 3\}$$
$$\text{COEND}$$
$$\{x = 3\}$$

Invariantes Globales

Se trata de expresiones definidas con las variables compartidas entre los procesos de un programa concurrente. Un Invariante Global (IG) se define como un predicado que captura la relación que existe entre las variables globales compartidas entre los procesos de un programa concurrente. También se puede usar para demostrar la *no interferencia*, ya que nos asegura que las demostraciones de los procesos están libres de interferencia si se puede escribir cualquier aserto C como una conjunción del tipo: $I \wedge L$. Donde I es un invariante global y L es un predicado en el que sóolo intervienen variables locales de un proceso y/o variables globales que sólo modifica el proceso a cuya demostración pertenece C. Los invariantes se utilizan para demostrar de una forma directa las propiedades de seguridad de los programas concurrentes, sin necesidad de tener que aplicar la regla de la concurrencia, cuya aplicación puede llegar a resultar muy tediosa en sistemas complejos con muchos procesos.

Para que un predicado I, definido a partir de las variables compartidas entre los procesos, pueda ser considerado un invariante global válido se han de cumplir las siguientes condiciones:

1. Es cierto para los valores iniciales de las variables.
2. Se mantiene cierto después de la ejecución de cada acción a, esto es, $\{I \wedge pre(a)\}\, a\, \{I\}$

Ejemplo de verificación utilizando un IG
Se trata de programar una transferencia entre 2 cuentas de un mismo banco como una transacción segura. Es decir, programar las secciones críticas necesarias para que un cliente pueda reintegrar de una cuenta e ingresar en otra sin que en ningun momento falte dinero del total constituido por los saldos de todas las cuentas. Además el sistema del banco ejecutará iterativamente y concurrentemente con las transferencias un programa para comprobar que la suma de los saldos se mantiene constante. Incluir las operaciones de sincronización necesarias para la comprobación de saldo constante y las operaciones en las cuentas, procurando optimizar la concurrencia total. Las acción atómica elemental `S1` representa la transacción formada por reintegro e ingreso de la cuenta ordenante a la beneficiaria, respectivamente.

`var c:array[1..n] of int;` cuentas
`{IG :: TOT=c[1]+...+c[n]= cte}`
la suma ha de ser constante (invariante global)
`P1::`
`A1:`$\{c[x] = X \wedge c[y] = Y\}$
`S1:`$< c[x] := c[x] - K;\ c[y] := c[y] + K >$
Transferencia de c[x] a c[y]
`A2:`$\{c[x] = X - K \wedge c[y] = y + K\}$

$IG :: \{Suma = c[1] + c[2] + ...c[i - c] \wedge i < n\} \wedge$
{Suma=c[1]+...+c[n]= cte}}

```
P2::
var Suma:=0; i:=1; Error:= false;
{Suma= c[1]+c[2]+ ... c[i-1]}
while i <= n do begin
  B1 : {Suma = c[1]+...c[i−1] ∧ i < n}
  S2 : {Suma := Suma + c[i]; }
  B2 : {Suma = c[1] + ...c[i − 1] + c[i]}
   i:= i+1;
  B3 : {Suma = c[1] + ...c[i − 1]}
end; enddo;
{Suma = c[1] + c[2] + ...c[n]}

if S<>TOT then Error:=true
```

Los asertos B1, B2 y B3 son críticos porque la evaluación de su valor lógico puede verse interferida por la asignación de los elementos del array "c" dentro de la acción atómica elemental

(S1) del proceso P1, que se ejecuta concurrentemente con P2. Los asertos A1, A2 no son críticos porque las instrucciones del proceso P2 sólo leen los valores de los elementos del array, pero no los modifican; por tanto, la evaluación del valor lógico de los asertos A1, A2 no puede sufrir interferencia en el fragmento de programa a verificar.

La acción atómica de P1 habría que sustituirla por una instrucción de sincronización:

$$\texttt{S1} :< \texttt{Espera}((\texttt{x} < \texttt{i} \wedge \texttt{y} < \texttt{i}) \vee (\texttt{x} > \texttt{i} \wedge \texttt{y} > \texttt{i})) \rightarrow \texttt{a[x]} := \texttt{a[x]} - \texttt{K};\ \texttt{a[y]} := \texttt{a[y]} + \texttt{K} >$$

El significado de la condición de sincronización anterior: la condición (x < i ∧ y < i) ∨ (x > i ∧ y > i) caracteriza el conjunto de estados en el que la suma de los valores del array "c" mantiene un valor constante y por tanto la ejecución de la acción atómica S1 no interfiere con los asertos críticos de la demostración anteriormente mostrada.

Exclusión de configuraciones

A veces resulta cómodo para demostrar propiedades de seguridad de sistemas concurrentes el formularlas como predicados que caracterizan estados del sistema que no pueden ser alcanzados por más de un proceso simultáneamente. Por ejemplo, en el problema de la exclusión mutua, es el caso en que las 2 precondiciones de acceso a las secciones críticas de los procesos `p1` y `p2` no pueden ser nunca ciertas a la vez. Si denominamos *NOSAFE* a un predicado que caracteriza un estado tal que los predicados P_1 y P_2 no pueden ser evaluados simultáneamente como ciertos, es decir: $NOSAFE = P_1 \wedge P_2 = FALSE$, entonces para demostrar que el programa P satisface la propiedad basta con probar que $NOSAFE = FALSE$ es un invariante global del programa. Formalmente, diríamos:

> **Demostración de la seguridad con un invariante**: sea *NOSAFE* un predicado que caracteriza un estado *no deseable* de un programa. Suponer que se puede demostrar el triple $\{P\}\, S\, \{Q\}$, y tal que $\{P\}$ caracteriza un estado inicial del programa, siendo I un invariante global de la demostración referida. Entonces S satisface la propiedad de seguridad especificada por $\neg\, NOSAFE$ si $I \Rightarrow \neg\, NOSAFE$.

```
var c:array[1..n] of int;
{TOT=c[1]+...+c[n]= cte}
P1::
A1:{c[x] = X ∧ c[y] = Y}
S1:< c[x] := c[x] − K; c[y] := c[y] + K >
A2:{c[x] = X − K ∧ c[y] = y + K}
```

```
P2::
var Suma:=0; i:=1; Error:= false;
{Suma= c[1]+c[2]+ ... c[i-1]}
while i <= n do begin
  B1 : {Suma = c[1]+...c[i−1] ∧ i < n}
  S2 : {Suma := Suma + c[i]; }
  B2 : {Suma = c[1] + ...c[i − 1] + c[i]}
   i:= i+1;
  B3 : {Suma = c[1] + ...c[i − 1]}
end; enddo;
{Suma = c[1] + c[2] + ...c[n]}

if S<>TOT then Error:=true
```

S_1 y S_2 se ejecutarán de forma mutuamente excluyente si:
$(\texttt{pre}(S_1) \wedge \texttt{pre}(S_2) = \texttt{NOSAFE}) \wedge \texttt{IR}_1 \wedge \ldots$
$\ldots \wedge \texttt{IR}_m = \texttt{FALSE}$
esto es:
$\{\texttt{c[x]} = \texttt{X} \wedge \texttt{c[y]} = \texttt{Y}\} \wedge$
$\{\texttt{Suma} = \texttt{c[1]} + \texttt{c[2]} + ...\texttt{c[i} - 1] \wedge \texttt{i} < \texttt{n}\} \wedge$
{Suma=c[1]+...+c[n]= cte}= FALSE}

Ejemplo de aplicación de IG

En el caso del ejemplo de las transferencias bancarias entre cuentas de un banco, se puede demostrar que el proceso `S1` (transferencia entre cuentas) y el `S2` (cálculo de saldo constante) se ejecutarán siempre en exclusión mutua. Para lo cual se utiliza el invariante global `Suma` y el predicado *NOSAFE*, es decir, se demuestra la imposibilidad de que los procesos evaluen simultáneamente como ciertas las precondiciones de `S1` y `S2` que representan los predicados A1 y A2, respectivamente.

1.5 Problemas resueltos

Ejercicio 1

Grafos de precedencia

1. Construir los programas concurrentes que se correspondan con los grafos de precedencia de la figura 1.9 utilizando el par `cobegin / coend`.

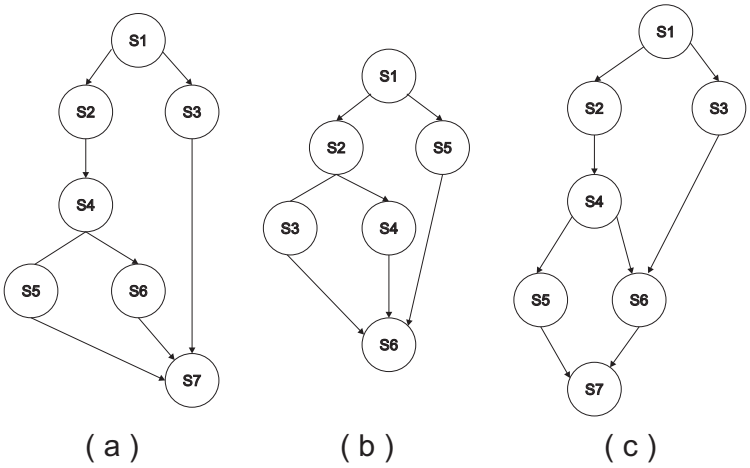

Figura 1.9: Grafos de precedencia

Solución:

```
(a)
S1
COBEGIN
    S3
    BEGIN
        S2; S4;
        COBEGIN
            S5; S6
        COEND
    END
COEND
S7
```

```
(b)
 S1;
 COBEGIN
     S5;
     BEGIN
         S2;
         COBEGIN
             S3;S4
         COEND
     END
     S6
 COEND
```

```
(c)
 // S5 espera a S3
 S1
 COBEGIN
     S3
     BEGIN S2;S4 END
 COEND
 COBEGIN
     S5;S6
 COEND
 S7
```

2. Dado el siguiente trozo de código obtener su grafo de precedencia correspondiente.

```
S0
COBEGIN
  S1
  BEGIN
    S2
    COBEGIN
      S3; S4
    COEND;
    S5
  END
  S6
COEND
S7
```

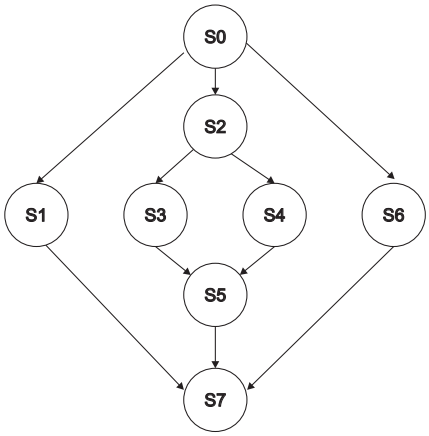

Figura 1.10: Solución

Ejercicio 2

Construir un programa que saque el máximo partido de la concurrencia para copiar un fichero secuencial `f` en otro fichero `g` utilizando la instrucción de creación de procesos concurrentes `cobegin / coend`.

Solución:

La solución al problema planteado sería la siguiente:

```
abrir_lectura(f)
abrir_escritura(g)
leer(f,r)
while (not fin(f)) do
    begin
        s:=r
        cobegin
            escribir(g,s)
            leer(f,r)
        coend
    end
    escribir(g,r)
end;
```

Ejercicio 3

Cinco filósofos dedican sus vidas a pensar y comer –estas acciones llevan un tiempo limitado. Los filósofos comparten una mesa rodeada de 5 sillas, cada una de éstas pertenece a un filósofo. En el centro de la mesa hay comida, y en la mesa hay 5 palillos y 5 platos. Cuando un filósofo no se relaciona con sus colegas, se supone que está pensando. De vez en cuando, un filósofo siente hambre y en ese caso se dirige a su silla y trata de coger los 2 palillos que están más cerca de él. Cuando un filósofo tiene sus 2 palillos simultáneamente, come sin dejarlos. Cuando ha terminado de comer, vuelve a dejar los 2 palillos y comienza a pensar de nuevo. Para solucionar el problema hay que inventar un *ritual* que permita comer a los filósofos, asegurando que no puede ocurrir la situación en que cada uno de los filósofos retenga 1 de los 2 palillos que le hacen falta para comer. Ya que si se diera esta situación ningún filósofo soltaría su palillo y, por tanto, ninguno llegaría a comer jamás. Programar la sincronización correcta, de acuerdo con la condición anterior, utilizando semáforos.

Solución:

La solución al problema planteado sería la siguiente:

```
semaphore_t palillo[5];
semaphore_t sitio;

inic(sitio,4);
for (i=0; i<5; i++) inic(palillo[i],1);

proceso filosof(int i){
   do{

       //piensa

       wait(sitio);
       wait(palillo[i]);
       wait(palillo[i+1]mod5);

       //come

       signal(palillo[i]);
       signal(palillo[i+1]mod5);
       signal(sitio);

   } while (true);
}
```

Ejercicio 4

Dos clases de procesos (procesos A y procesos B) entran en una habitación. Un proceso A no puede salir hasta que no haya en la habitación 2 procesos del tipo B y 1 proceso B no puede salir hasta que no haya en la habitación 1 proceso A. Programar esta sincronización utilizando semáforos.

Solución:

La solución al problema planteado sería la siguiente:

```
semaforo presenA, presenB, S;
inic(presenA,0);
inic(presenB,0);
inic(S,1);
```

```
Proceso A
---------
wait(S);
do{
    if(!salida_iniciada){
        nA++;
        while(nB<2){
            signal(S);
            wait(presenB);
            wait(S);
        }
        if(!salida_iniciada){
            salida_iniciada=true;
            signal(presenA);
            signal(S);
            break;
        }
    }
} while (true);
wait(S);
if(salida_iniciada){
    nA--;
    nB-=2;
    salida_iniciada=false;
}
signal(S);
```

```
Proceso B
---------
wait(S);
do{
   if(!salida_iniciada){
     nB++;
     while(nA==0||nB<2){
        signal(S);
        wait(presenA);
        wait(S);
     }
     if(nA>=1 && nB==2) signal(presenA);
     if(!salida_iniciada){
          salida_iniciada=true;
          signal(presenB);
          signal(S);
          break;
     }
   }
} while(true);
wait(S);
if(salida_iniciada){
  nA--;
  nB-=2;
  salida_iniciada=false;
}
signal(S);
```

Ejercicio 5

Programar utilizando semáforos 2 procesos que leen datos de 1 fichero continuamente, que por cada dato que leen van a comprobar si está en una tabla (leen de la tabla) y si está lo marcan (se escribe en la tabla). Considerar que un proceso acaba cuando llega al final del fichero, que el acceso al fichero se realiza en exclusión mutua, que en la tabla pueden leer los 2 procesos a la vez, pero que cuando uno escribe no puede estar el otro trabajando sobre la tabla, ni leyendo ni escribiendo.

Solución:

La solución al problema planteado sería la siguiente:

```
int nl, // numero lectores leyendo
    ne, // numero escritores escribiendo
    nle, // numero lectores esperando
    nee; // numero escritores esperando

semaforo leer, escribir, S;

--------

inic(leer,0);
inic(escribir,0);
inic(S,1);

--------

comenzar_leer(){
    wait(leer);
    wait(S);
    nl++;
    if(nl==1)wait(escribir);
    signal(S);
    signal(leer);
}

fin_leer(){
    wait(S);
    nl--;
    if(nl==0) signal(escribir);
    signal(S);
}
```

```
comenzar_escribir(){
    wait(leer);
    wait(escribir);
}

fin_escribir(){
    signal(escribir);
    signal(leer);
}
```

```
Proceso Pi
----------
abrir_lectura(f);
wait(s)
leer(fato)
signal(s)
while(!EOF(f)){
    comenzar_leer();
    if(leido){
        fin_leer()
        comenzar_escribir()
        escribir()
        fin_escribir()
    } else {
       fin_leer
    }
    wait(s)
    leer()
    signal(S)
}
```

1.6 Problemas propuestos

1. Considerar el siguiente fragmento de programa para 2 procesos:

```
int x;

process A {
  int i;
  for(i=0; i<10; i++)
   x= x + 1;
}
process B{
  int j;
  for (j=0; j<10;j++)
   x= x + 1;
}
```

```
main(){
x=0;
cobegin
   A;
   B;
coend;
}
```

   Suponiendo que los 2 procesos pueden ejecutarse a cualquier velocidad relativa –el uno respecto del otro. ¿Qué valor tendrá la variable x al terminar el programa? Suponer que la instrucción `x = x + 1;` da lugar a las 3 intrucciones siguientes de un lenguaje ensamblador cualquiera:

   (a) `LOAD X R` (*cargar desde memoria el valor de la variable* **x** *en el registro R*)

   (b) `ADD R 1` (*incrementar el valor acumulado en R*)

   (c) `STORE R X`(*almacenar el contenido de R en la posición de memoria de* **x**)

2. ¿ En qué consiste exactamente la ventaja de la concurrencia en los sistemas monoprocesador?

3. ¿Qué se entiende por *error transitorio* en las secuencias de ejecución de los programas concurrentes?

4. ¿A qué hace referencia el término *condición de carrera* entre procesos concurrentes?

5. ¿ Cuáles son las diferencias entre programación concurrente, paralela y distribuida?

6. Supongamos que disponemos de un contador de energía eléctrica que genera impulsos cada vez que consume 1 Kw de energía y queremos implementar un sistema que cuente el número de impulsos generados en 1 hora, es decir, que cuente el número de Kw consumidos a la hora. Para ello el sistema estará formado por 2 procesos:

   - Un proceso acumulador que lleva la cuenta de los impulsos recibidos.
   - Un proceso escritor que escribe en la impresora.

   En una variable (n) se lleva la cuenta de los impulsos. Si el sistema se encuentra en un estado correspondiente al valor de la variable (n= N)y, en esas condiciones, se presentan simultáneamente un nuevo impulso y el final del periodo de 1 hora, obtener las posibles secuencias de ejecución de los procesos y cuáles de ellas son correctas.

7. En el siguiente programa se espera que como resultado de su ejecución se imprima `110` ó `50` ¿ Es correcto el siguiente código?

```
int x;
process P1{
  x += 10;
}
process P2{
  if(x >100)
     write(x);
     else write (x-50);
 }

main(){
  x= 100;
  cobegin  P1; P2 coend;
}
```

8. Supongamos que 2 procesos concurrentes `P1` y `P2` intentan usar el mismo dispositivo de E/S. De manera más concreta supongamos que `P1` intenta leer del bloque 20 del dispositivo, mientras que `P2` intenta escribir en el bloque 88. Si las operaciones de leer y escribir un bloque son atómicas, la ejecución de `P1` y `P2` no produciría problemas, pero si estas operaciones se pueden descomponer en las básicas de saltar al bloque correspondiente y leer o escribir, como se muestra más abajo, se podrían producir errores. ¿ Qué secuencias de ejecución, entre las posibles para `P1` y `P2`, son correctas y cuáles no los son?

```
read(20)<->  saltar(20), read()          write(88)<->  saltar(88), write()
```

9. Supongamos un sistema multiprocesador que posee una instrucción llamada *Test and Set()* (T&S). De tal forma que ejecutando `T&S(x)` se obtendría lo mismo que ejecutando las 2 sentencias siguientes:

```
x= c;
c= 1;
```

   Donde `x` es una variable local y `c` es una variable global del sistema. Utilizando la instrucción `T&S`, construir los protocolos de adquisición, de restitución y la inicialización de las variables para dar una solución al problema del acceso en exclusión mutua a una sección crítica compartida por 2 procesos de un programa concurrente.

10. Suponer que el siguiente algoritmo (inserción directa) quisiese ser utilizado en un programa paralelo de la forma siguiente:

```
cobegin sort(1,n); sort(n+1, 2*n); coend; merge(1, n+1, 2*n);
procedure sort(inferior, superior: int);
var i,j:int;
begin
  for i:=inferior to superior-1 do
    for j:= i+1 to superior do
       if (a[j]<a[i]) then swap(a[i],a[j]);
end;
```

    - Si el tiempo de ejecución secuencial del algoritmo es $t(n) \equiv \frac{n}{2}$ ¿Cuál sería el tiempo de ejecución del programa paralelo si la mezcla lleva n operaciones en realizarse?
    - Escribir un procedimiento de mezcla que permita paralelizar las 3 operaciones:

      `cobegin sort(1,n); sort(n + 1, 2 * n); merge(1,n + 1,2 * n); coend`

11. Sea una sucesión en la que cada término es la suma de los 2 anteriores:

$$\texttt{a(1)} = 0;\ \texttt{a(2)} = 1;\ \ldots\ \texttt{a(i)} = \texttt{a(i} - 1) + \texttt{a(i} - 2)$$

    Para calucular las sumas parciales de los términos impares de esta sucesión, tenemos 2 procesos: *sucesión* y *suma*. El primero calcula los términos de la sucesión y el segundo calcula las sumas parciales de los términos impares. Implementar el algoritmo concurrente anterior incluyendo únicamente operaciones sobre los semáforos.

12. Si se desea pasar un semáforo como parámetro a un procedimiento:
    - ¿Cómo habría que pasarlo, por valor, o por referencia, o bien no hay ninguna diferencia?
    - ¿Qué ocurre si se toma la opción equivocada?

13. ¿Cómo se puede saber en un programa concurrente el número de procesos que están bloqueados en un semáforo?

14. Utilizando sólo semáforos binarios, construir 2 funciones `genwait()` y `gensignal()` para simular los semáforos generales.

15. Demostrar que si las operaciones de los semáforos `wait` y `signal` no se ejecutan de forma atómica, no se verifica la exclusión mútua que proporciona el mecanismo de semáforo.

16. Añadir un semáforo al siguiente programa para que siempre escriba 40

```
//program incrementar;
int n;
process inc(){
 int i;
 for(i=0; i<20;i++)
   n++;
 }
main(){
  n= 0;
  cobegin
    inc();inc();
  coend;
  write(n);
}
```

17. Suponer que para fumar un cigarillo hacen falta 3 ingredientes: (a) tabaco, (b) papel y (c) cerillas). Suponer también tres fumadores sentados alrededor de una mesa, cada uno de los cuales tiene un suministro ilimitado (nunca consigue acabarlo) de uno de los ingredientes – un fumador tiene siempre tabaco, otro papel, y el tercero tiene cerillas. Suponer también que hay un árbitro que no es fumador, que permite a cada fumador liar sus cigarrillos, para lo cual selecciona a 2 fumadores arbitrariamente (*no determinísticamente*), les quita uno de los ingredientes que poseen cada uno y los coloca encima de la mesa. Entonces, el árbitro notifica al tercer fumador que los 2 ingredientes que le faltan para liar están ya encima de la mesa. Este coge los ingredientes y los utiliza (junto con el ingrediente que el posee) para liarse un cigarrillo, que tardará en fumar un rato. Mientras, el árbitro,

viendo la mesa vacía, vuelve a elegir 2 fumadores al azar y les quita 1 ingrediente a cada uno, depositándolos en la mesa. Este proceso continua repetidamente. Un fumador sólo puede comenzar a liarse otro cigarrilo cuando ha terminado de fumarse el último que lió. Los fumadores no pueden reservarse ingredientes que aparecen en la mesa; si el árbitro coloca tabaco y papel mientras que el fumador de las cerillas está fumando, el tabaco y el papel permanecerán en la mesa hasta que el fumador de las cerillas acabe su cigarrillo y recoga de la mesa estos ingredientes. Programar la sincronización entre el árbitro y los fumadores utilizando semáforos.

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 en el que se cumplan las siguientes restricciones:
    - Chicas, sólo puede haber 1 en el servicio.
    - Chicos, puede haber más de 1, pero como máximo se admitirán a 5.
    - (Versión machista) los chicos tienen preferencia sobre las chicas. Esto quiere decir que si una chica está esperando y llega un chico, este puede pasar. Incluso si no pudiera entrar inmediatamente porque hay 5 chicos dentro del servicio, pasará antes que la chica cuando salga algún chico.
    - (Versión feminista) 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 no puede entrar en el servicio porque ya hay una chica dentro, pasará antes que el chico cuando salga la chica.

19. Construir un grafo de sincronización que establezca precedencias entre los procesos S_i para que el siguiente código pueda ser ejecutado por los procesos de forma concurrente y sin problemas de interferencia. Para resolverlo se han de encontrar conjuntos de instrucciones entrelazables que no ocasionen condiciones de carrera en el acceso a las variables. De ahí se obtendrían los procesos que se pueden lanzar concurrentemente dentro de pares `cobegin / coend`:

```
S1:: cuad = x*x;
S2:: m1= a*cuad;
S3:: m2= b*x;
S4:: z= m1+m2;
S5:: y= z+c;
```

20. ¿Qué se entiende por un programa concurrente correcto?

21. ¿No es contradictorio el hecho de que los procesos de un programa concurrente puedan sufrir inanición con el requisito que dice que los procesos necesariamente han de avanzar en la ejecución de sus instrucciones (*Hipótesis de progreso finito*)? Justificar la respuesta.

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

```
a) { i<10 } i:= 2*i+1; {          }
b) { i>0 } i:= i-1; {         }
c) { i>j }  i:= i+1; j:= j+1; {    }
d) {F } a:= a+7; {  }
e) {   } i:=3; j:= 2*i;  {          }
f) {   } c:=a+b; c:= c/2;
          {                         }
```

23. Explicar las diferencias entre las tres condiciones siguientes referidas al grado de equidad en la asignación del procesador a los procesos de un programa concurrente (se supone planificación por quantum de tiempo)

- Equidad incondicional o imparcialidad : asegura que un proceso activo conseguirá ver ejecutadas sus instrucciones muy a menudo.
- Equidad débil o justicia: asegura que un proceso en cuyo código se han programado instrucciones que dependen del valor de verdad de condiciones, dichas instrucciones serán ejecutadas muy a menudo si el valor de verdad de dichas condiciones se mantiene.
- Equidad fuerte: si las condiciones de las que dependen la ejecución de determinadas instrucciones alcanzan el valor de verdad favorable muy amenudo, entonces dichas instrucciones se ejecutarán también muy a menudo.

24. Indicar con qué tipo de equidad un planificador de procesos aseguraría que el siguiente programa de 2 procesos concurrentes siempre finalizaría en un tiempo no arbitrariamente grande:

```
bool continuar=TRUE;
sem_t intentar;
sem_init(&intentar, 0, 1);
sem_wait(&intentar);
```

```
cobegin
S1:: while (continuar) {
       sem_post(&intentar);
       if (continuar)
          sem_wait(&intentar);
      }
S2::  sem_wait(&intentar);
        continuar= FALSE;
coend;
```

25. Dado el programa:

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

(a) ¿ Cuáles son los posibles valores de `x` y de `y`?

(b) ¿ Cuáles serían los posibles valores de `x` y de `y` en el caso de que se suprimieran los ángulos y cada orden de asignación fuera implementada usando tres instrucciones atómicas: lectura de memoria, suma o multiplicación y escritura de registro a memoria?

26. Comprobar si `{x`$\geq$ `2}` `<x=x-2>` interfiere con los términos:

(a) `{x`$\geq$ `0}` `<x=x+3>` {`x` $\geq$ `3`}

(b) `{x`$\geq$ `0}` `<x=x+3>` {`x` $\geq$ `0`}

(c) `{x`$\geq$ `7}` `<x=x+3>` {`x` $\geq$ `10`}

(d) `{y`$\geq$ `0}` `<y=y+3>` {`y` $\geq$ `3`}

(e) `{xes impar}` `<y=x+3>` {`y es par`}

27. Estudiar cuáles son los valores finales de las variables x e y en el siguiente programa. Insertar asertos entre las llaves que aparecen antes y después de cada sentencia para crear

una traza de demostración del citado programa.

```
int x=C1;
int y=C2;
x=x+y;
y=x*y;
x=x-y;
```

Traza de la demostración:

```
        {               }
             x=x+y;
        {               }
             y=x*y;
{                            }
             x=x-y;
{                            }
```

28. Demostrar que el siguiente triple es cierto:

```
                {x==0}
                cobegin
<x=x+1> || <x=x+2> || <x=x+4>
                coend
                {x==7}
```

29. Dada la siguiente construcción de composición concurrente P:

```
                P::cobegin
<x=x-1>;<x=x+1>; || <y=y+1>;<y=y-1>;
                  coend
```

demostrar que {x==y} P {x==y}.