libro.pdf

FaReLiLoCa

Sistemas Concurrentes y Distribuidos

3º Doble Grado en Ingeniería Informática y Administración y Dirección de Empresas

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación
Universidad de Granada

Sistemas Concurrentes y Distribuidos
Teoría y Práctica
Tomo I - Teoría

Manuel I. Capel Tuñón
Sandra Rodríguez Valenzuela

Contenidos

Sistemas Concurrentes y Dist...

Banco de apuntes de la

WUOLAH

Comparte estos flyers en tu clase y consigue más dinero y recompensas

1. Imprime esta hoja
2. Recorta por la mitad
3. Coloca en un lugar visible para que tus compis puedan escanar y acceder a apuntes
4. Llévate dinero por cada descarga de los documentos descargados a través de tu QR

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.

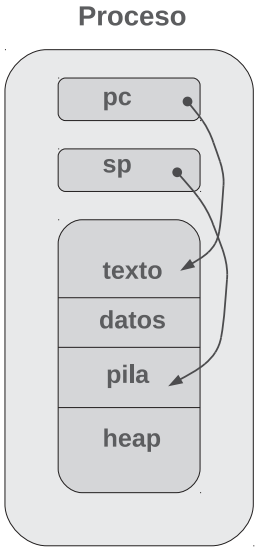


Figura 1.1: Representación de un proceso

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

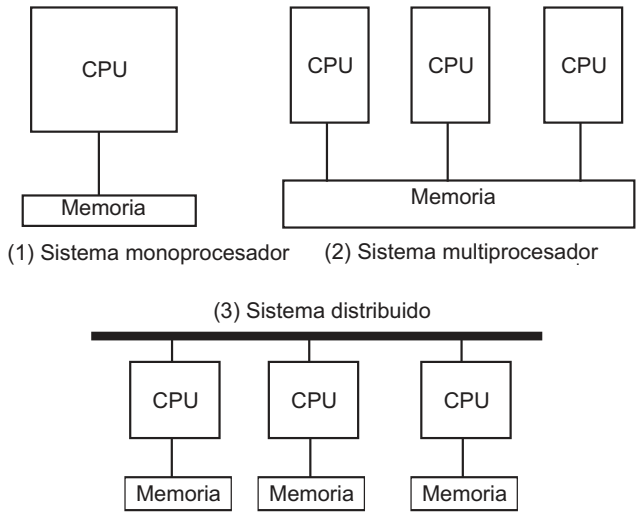


Figura 1.5: Arquitecturas de sistemas con diferentes grados de paralelismo

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.

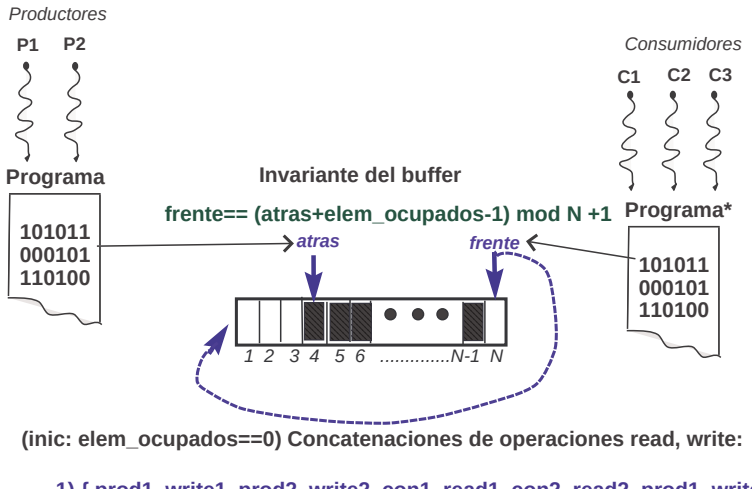


Figura 1.6: Representación del funcionamiento del buffer en un *productor-consumidor*

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

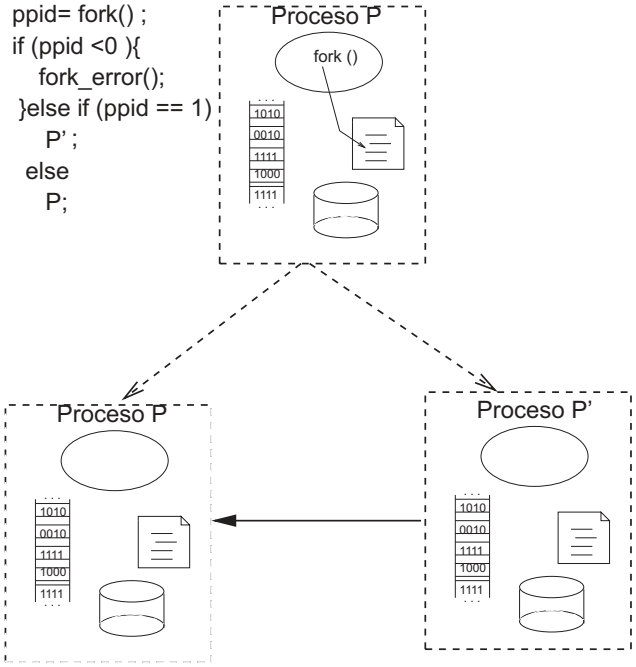


Figura 1.7: Operación fork() en UNIX

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:

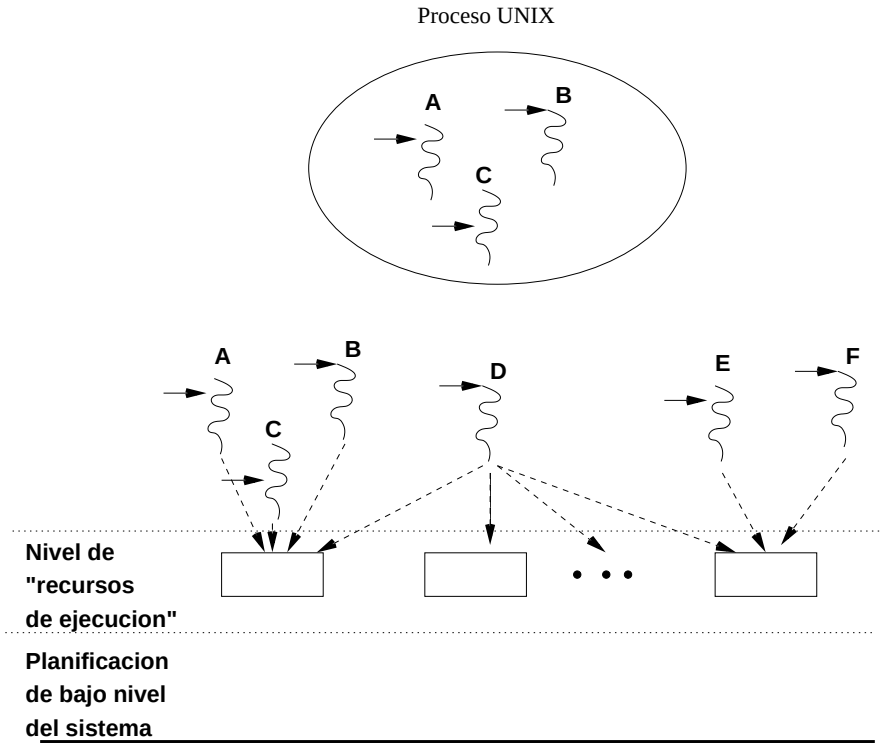


Figura 1.8: Representación de planificación de hebras POSIX 1003.1c (thread package)

- 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:

$$asertos \subseteq hechos$$

  Sí̃lo 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:
    - $\neg(\mathrm{P} \wedge \mathrm{Q}) = \neg\mathrm{P} \vee \neg\mathrm{Q}$
    - $\neg(\mathrm{P} \vee \mathrm{Q}) = \neg\mathrm{P} \wedge \neg\mathrm{Q}$
12. Eliminación-*And*: $(\mathrm{P} \wedge \mathrm{Q}) \rightarrow \mathrm{P}$
13. Eliminación-*Or*: $\mathrm{P} \rightarrow (\mathrm{P} \vee \mathrm{Q})$

La regla 12 se puede explicar diciendo que el conjunto de estados que cumplen P incluye al conjunto de los que cumplen $\mathrm{P} \wedge \mathrm{Q}$; se dice que la proposición P es más débil y, por tanto, se *ve implicada* por la expresión (*más fuerte*) $\mathrm{P} \wedge \mathrm{Q}$. 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 $\{\mathrm{P}\}\,\mathrm{S}\,\{\mathrm{Q}\}$; 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

$\{\mathrm{P}\}\,\mathrm{S}\,\{\mathrm{Q}\}$ 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 $\{\mathrm{P}\}\,\mathrm{null}\,\{\mathrm{P}\}$: 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:
$(\texttt{pre}(S_1) \wedge \texttt{pre}(S_2) = \texttt{NOSAFE}) \wedge IR_1 \wedge \ldots$
$\ldots \wedge IR_m = \texttt{FALSE}$
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`.

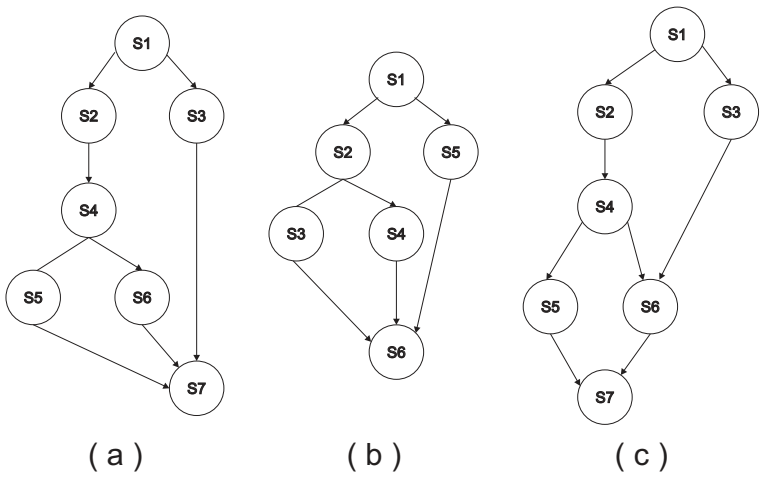


Figura 1.9: Grafos de precedencia

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
```

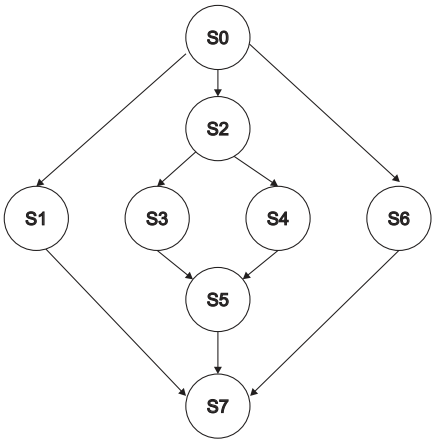


Figura 1.10: Solución

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:

      $$\texttt{cobegin sort(1,n); sort(n+1, 2*n); merge(1,n+1,2*n); coend}$$

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≥ 2} <x=x-2> interfiere con los términos:
    (a) {x≥ 0} <x=x+3> {x ≥ 3}
    (b) {x≥ 0} <x=x+3> {x ≥ 0}
    (c) {x≥ 7} <x=x+3> {x ≥ 10}
    (d) {y≥ 0} <y=y+3> {y ≥ 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}.

Fuentes Consultadas

[Andrews, 1991] Andrews, G. (1991). *Concurrent programming: principles and practice*. Benjamin Cummings, Redwood City, California.

[Andrews, 1999] Andrews, G. (1999). *Foundations of Multithreaded, Parallel, and Distributed Programming*. Benjamin Cummings, Redwood City, California.

[Axford, 1989] Axford, T. (1989). *Concurrent Programming: Fundamnetal Techniques for Real–Time and Parallel Software Design*. John Wiley, Chichester (UK).

[Barnes, 1994] Barnes, J. (1994). *Programming in Ada. Plus an Overview of Ada 9X*. Addison-Wesley, New-York.

[Ben-Ari, 2006] Ben-Ari, M. (2006). *Principles of Concurrent and Distributed Programming*. 2nd Edition. Addison-Wesley.

[Brinch-Hansen, 1975] Brinch-Hansen, P. (1975). The programming language concurrent pascal. *IEEE Transactions on Software Engineering*, 1(2):199–207.

[Butenhof, 1997] Butenhof, D. (1997). *Programming with POSIX threads*. Addison-Wesley.

[Dijkstra, 1965] Dijkstra, Edsger W. (1965) *Cooperating sequential processes*. Center for American History, University of Texas at Austin.

[Gallmeister, 1995] Gallmeister, B. (1995). *Programming for the real world: POSIX 4.0*. O'Reilly, Sebastopol, California.

[INMOS, 1984] INMOS (1984). *Occam Programming Manual*. Prentice-Hall, USA.

[Lea, 2001] Lea, D. (2001). *Programación concurrente en Java: principios y patrones de diseño*. Addison-Wesley/Pearson Education.

[Snir et al., 1999] Snir, M., Otto, S., Huss-Lederman, S., and D. Walker, J. D. (1999). *MPI: The Complete Reference*. The MIT Press, Cambridge, USA.

[Wirth, 1985] Wirth, N. (1985). *Programming in Modula-2*. Springer Verlag, Berlin.

Capítulo 2

Algoritmos y Mecanismos de Sincronización Basados en Memoria Compartida

2.1 Introducción al problema de acceso a una sección crítica en exclusión mutua por parte de los procesos

Siempre han existido soluciones de bajo nivel para resolver el problema de acceso a una sección crítica en exclusión mutua, p.e. los cerrojos; sin embargo, se han venido estudiando desde 1965 una serie de soluciones-software para resolverlo. Dichas soluciones pretenden ser independientes de las instrucciones de una máquina concreta y permiten asegurar que los procesos cumplen todas las propiedades exigidas a un programa concurrente. Se intentan conseguir, fundamentalmente, soluciones que proporcionen un acceso equitativo de los procesos a la sección crítica. Sólo vamos a considerar, por ahora, soluciones al problema de la exclusión mutua que utilicen las instrucciones básicas de lectura o escritura del valor de una variable.

Los algoritmos que se van a introducir a continuación representan una muestra reducida, y procurando respetar el orden cronológico de aparición, de las propuestas de solución del problema de la exclusión mutua. Dichos algoritmos poseen un indudable valor pedagógico para la comprensión de conceptos básicos de la Programación Concurrente.

Los algoritmos para resolver el problema de la exclusión mutua suelen ser categorizados en dos grupos:

1. *Centralizados:* utilizan variables compartidas entre los procesos, que expresan el estado de estos. A estas variables se suele acceder en secciones diferenciadas del código de los procesos, denominadas protocolos de *adquisición* y *restitución*, que han de ejecutar los procesos concurrentes antes y después de la sección crítica.

2. *Distribuidos:* no utilizan variables compartidas entre los procesos. Suelen utilizar sólo instrucciones de paso de mensajes para detener a los procesos cuando la sección crítica está ocupada o para *informar* que vuelve a estar libre.

2.1.1 Solución al problema con bucles de espera ociosa

La espera ociosa consiste en que los procesos realizan iteraciones de un bucle vacío, antes de entrar a la sección crítica, hasta que dicha entrada sea *segura*. Este tipo de implementación utiliza recursos (tiempo del procesador en sistemas monoprocesador y ciclos de memoria en sistemas multiprocesador) sin realizar ningún avance significativo en la ejecución del proceso que espera. La espera ociosa se puede considerar una solución aceptable al problema del acceso en exclusión mutua a una sección crítica siempre que el sistema no tenga muchos procesos.

2.1.2 Condiciones que ha de verificar una solución correcta al problema

El problema del acceso a la sección crítica en exclusión mutua sólo utilizando bucles de espera ociosa y valores de variables compartidas para 2 procesos dio lugar a muchas propuestas que no resultaban fáciles de evaluar. Egbert Dijkstra [Dijkstra, 1965] estableció un método para obtener un algoritmo que permitiera encontrar una solución al problema y las condiciones que debería cumplir para ser aceptado como tal. Las condiciones de Dijkstra son las siguientes:

1. *No hacer ninguna suposición acerca de las instrucciones o número de procesos soportados por la máquina.* Sólo podemos utilizar las instrucciones básicas, tales como leer, escribir o comprobar una posición de memoria para resolver el problema. Las instrucciones mencionadas se ejecutan sin interrupción, *atómicamente*, es decir, en caso de que 2 ó más instrucciones fueran susceptibles de ser ejecutadas simultáneamente por los procesos del programa, el resultado de dicha ejecución es *no-determista*: equivalente a la ejecución secuencial de dichas instrucciones en un orden no predecible.

2. *No hacer ninguna suposición acerca de la velocidad de ejecución de los procesos, excepto que no es cero.*

3. *Cuando un proceso está en sección no-crítica* no puede impedir a otro que entre en ella.

4. *La sección crítica será alcanzada por alguno de los procesos que intentan entrar.* Esta condición asegura siempre la alcanzabilidad de la sección crítica por parte de los procesos. Excluye la posibilidad de que en una ejecución se llegara a un interbloqueo, es decir, que ningún proceso consiguiera jamás entrar en sección crítica, pero no asegura que todos los procesos del protocolo consigan alguna vez entrar o que consigan hacerlo de forma *equitativa.*

2.2 Método de refinamiento sucesivo

Además de las condiciones que ha de cumplir un algoritmo para ser considerado una solución al problema de la exclusión mutua, Dijkstra propone un método para obtener el algoritmo en 4 pasos o *modificaciones* de un esquema inicial. Inicialmente se supone que los procesos alternan su entrada a la sección crítica según el valor de una variable global compartida entre ellos, que asigna el *turno*. Cada una de las citadas *modificaciones* se considera una *etapa* en el proceso de

encontrar la solución aceptable al problema de la exclusión mutua, que finalmente se obtendrá en la quinta etapa y que coincide con el denominado algoritmo de Dekker.

Primera etapa:
Se utiliza una variable `turno` que contiene el identificador del proceso que puede entrar en sección crítica. Dicha variable puede valer 1 ó 2, inicialmente suponemos que su valor es 1.

```
   P1                      P2
---------               ---------
while true do begin      while true do begin
  Resto ;                  Resto;
  while turno <> 1 do      while turno <> 2 do
     nothing;                 nothing;
  enddo;                   enddo;
  S.C.                     S.C.
  turno:= 2;               turno:= 1;
 end                      end
enddo;                   enddo;
```

La solución garantiza el acceso en exclusión mutua de los procesos, esto es, la solución es segura. No garantiza, sin embargo, la tercera condición de Dijkstra, ya que los procesos sólo pueden entrar en la sección crítica alternativamente. Podría pensarse que la solución para evitar la alternancia forzosa en el acceso a la sección crítica consistiría en asignar `turno:= i` antes del bucle de espera ociosa de cada proceso Pi; pero, resulta muy fácil de comprobar que la solución dejaría de ser segura en ese caso.

Segunda etapa:
La alternancia estricta en el acceso a la sección crítica que se producía en la primera etapa era debida a que, para decidir qué proceso entraba en sección crítica, se tenía que almacenar información global del estado del programa en una variable `turno` que sólo la cambiaba un proceso al salir de esta. Para evitarlo, la idea ahora es asociar con cada proceso su información de estado en una variable *clave* que indique si dicho proceso está en sección crítica o no. Según este esquema, ahora cada uno de los procesos lee la clave del otro pero no puede modificarla.

```
Inicialmente: c1=c2= 1;
   P1                      P2
---------               ---------
while true do begin      while true do begin
  Resto ;                  Resto;
  while c2=0 do            while c1=0 do
    nothing;                 nothing;
  enddo;                   enddo;
  c1:= 0;                  c2:= 0;
    S.C.                     S.C.
  c1:= 1;                  c2:= 1;
  end                      end
enddo;                   enddo;
```

En este caso falla la propiedad de seguridad. Ya que si P1 y P2 se ejecutan a la misma velocidad, comprueban que el otro proceso no está en sección crítica y ambos pueden entrar en sección crítica. Cuando asignan sus claves a 0 ya es demasiado tarde, pues ya han conseguido pasar el bucle de espera. Como esta solución sólo funcionará dependiendo de la velocidad de los procesos, se dice que no cumple la segunda condición de Dijkstra.

Tercera etapa:
El problema de la solución anterior consiste en que un proceso podría estar comprobando el estado del otro al mismo tiempo que este lo modifica. Esto se debe a que la salida de un proceso de su bucle de espera ociosa, y la asignación de su clave a cero, no se realiza como una única operación indivisible, es decir, *atómicamente*. Por tanto, un proceso podría resultar desplazado del procesador después de salir de su bucle de espera, pero antes de cambiar el valor de su clave. Si el otro proceso, a continuación, se convirtiera en activo no detectaría el nuevo valor de la clave del primero y también saldría de su bucle de espera activa, llegándose a una situación en que ambos entrarían a la sección crítica.

La solución que ahora se propone consiste en cambiar la posición de la sentencia de asignación de la clave del proceso antes del bucle de espera ociosa. De esta forma, sería imposible que un proceso pase dicho bucle con un valor de su clave distinto de cero, evitándose, por tanto, el posible escenario de falta de seguridad anteriormente comentado en la segunda etapa del método.

```
Inicialmente: c1=c2= 1;
       P1                        P2
    ---------                 ---------
    while true do begin       while true do begin
      Resto ;                    Resto;
      c1:= 0;                    c2:= 0;
      while c2=0 do              while c1=0 do
        nothing;                   nothing;
      enddo;                     enddo;
          S.C.                       S.C.
      c1:= 1;                    c2:= 1;
      end                        end
     enddo;                     enddo;
```

Esta solución asegura el acceso en exclusión mutua, pero no se puede considerar correcta, ya que si ambos procesos tienen la misma velocidad se podría producir una situación de interbloqueo, podrían cambiar sus claves a cero y, a continuación, bloquearse realizando iteraciones de los bucles indefinidamente. Por lo tanto, decimos que este protocolo es inaceptable porque no se cumple la cuarta condición de Dijkstra.

Cuarta etapa:
Lo que causa el problema de la tercera etapa es que cuando un proceso modifica el valor de su clave no sabe si el otro proceso está haciendo lo mismo, concurrentemente con él.

La solución ahora sería permitir a un proceso volver a cambiar el valor de su clave a `1` si, despues de asignar su clave a `0`, comprueba que el otro proceso también cambió su clave al valor cero.

```
Inicialmente: c1=c2= 1;
       P1                        P2
    ---------                 ---------
    while true do begin        while true do begin
      Resto ;                     Resto;
      c1:= 0;                     c2:= 0;
      while c2=0 do               while c1=0 do
       begin                       begin
         c1:= 1;                     c2:= 1;
         while c2=0 do               while c1=0 do
           nothing;                    nothing;
         enddo;                      enddo;
         c1:=0;                      c2:= 0;
       end                         end
       enddo;                      enddo;
         S.C.                        S.C.
       c1:= 1;                     c2:= 1;
    end                          end;
   enddo;                       enddo;
```

Si ambos procesos se ejecutasen a la misma velocidad se podría seguir produciendo interbloqueo, aunque ahora es más improbable que se produzca que en la tercera etapa. Por lo tanto, con esta solución seguirían sin cumplirse ni la segunda ni la cuarta condición de Dijkstra.

Las variables c1 y c2 pueden interpretarse como unas variables de estado de los procesos, que informan si el proceso está intentando entrar en sección crítica o no. La conclusión a la que se llega después de los intentos anteriores es que el leer el valor de las variables de estado de los procesos no es suficiente para dar una solución correcta al problema de la exclusión mutua. Es necesario, por tanto, utilizar un orden previamente establecido para entrar en sección crítica si hay conflicto entre los procesos. Dicho orden lo establece una variable `turno`.

2.2.1 Algoritmo de Dekker

El algoritmo de Dekker se basa en la primera y cuarta etapas del método de Dijkstra. La primera etapa era segura pero presentaba el problema de que la variable `turno` era global y no podía ser cambiada por un proceso antes de entrar en su sección, aunque el otro proceso estuviera pasivo, lo que llevaba a la alternancia estricta en el acceso a la sección crítica por parte de los procesos y, como consecuencia de esto, no se cumplía la tercera condición. Por otra parte, la cuarta etapa, basada en claves separadas que representan el estado de avance de los procesos respecto de la ejecución del protocolo no cumple la cuarta condición de Dijkstra.

En el algoritmo de Dekker, un proceso que intenta entrar en sección crítica asigna su clave a cero. Si encuentra que la clave del otro es también cero, entonces consulta el valor de `turno`; si posee el turno, entonces insiste y comprueba sucesivamente la clave del otro proceso. Eventualmente el otro proceso, cambiando su clave a 1, le cede el turno y este consigue finalmente entrar en sección crítica.

```
   P1                          P2
---------                   ---------
while true do begin         while true do begin
  Resto ;                     Resto;
  c1:= 0;                     c2:= 0;
  while c2=0 do               while c1=0 do
   if turno= 2 then            if turno= 1 then
     begin                       begin
       c1:= 1;                     c2:= 1;
       while turno= 2 do           while turno=1 do
          nothing;                    nothing;
       enddo;                      enddo;
       c1:=0;                      c2:= 0;
     end                         end
    endif                       endif
   enddo;                      enddo;
     S.C.                        S.C.
   turno:= 2;                  turno:= 1;
   c1:= 1;                     c2:= 1;
 end;                        end;
enddo;                      enddo;
```

2.2.2 Verificación de propiedades

Exclusión mutua:
El proceso `Pi` entra en sección crítica sólo si `cj=1`. La clave de un proceso sólo la puede modificar el propio proceso. El proceso `Pi` comprueba la clave del otro proceso `cj` sólo después de asignar su clave `ci= 0`. Luego, cuando el proceso `Pi` entra, y se mantiene, en sección crítica se cumple que `ci= 0` $\wedge$ `cj=1`. Los valores de estas variables indican que sólo un proceso puede acceder a sección crítica cada vez.

Alcanzabilidad de la sección crítica:
Esquema de la demostración:

1. Si suponemos que un sólo proceso `Pi` intenta entrar en sección crítica, encontrará la clave del otro proceso con valor `cj=1`; por tanto, el proceso `Pi` puede entrar.

2. Sin embargo, si los procesos `Pi`, `Pj` intentan entrar en sección crítica y el `turno=i`, entonces se tendrá el siguiente escenario:

   (a) `Pj` encuentra la clave `ci=1`, entonces `Pj` entra en sección crítica;

   (b) `Pj` encuentra la clave `ci=0` y `turno=i`, entonces se detiene en su bucle interno y pone `cj=1`;
   `Pi` encuentra la clave `cj= 0`, se detiene en el bucle externo hasta que `cj= 1`;
   por último `Pi` entra en sección crítica.

Posible inanición de un proceso:
En el caso de que `P2`, por ejemplo, fuera un proceso muy rápido y repetitivo, entonces podría estar entrando contínuamente en sección crítica e impidiendo al proceso `P1` el hacerlo. El escenario consistiría en que `P1` sale de su bucle interior, pero nunca puede escribir el valor de su clave c1, ya que P2 siempre la está leyendo y se lo impide.

Equidad del protocolo:
La equidad del algoritmo del Dekker dependerá de la equidad del hardware. Si existen peticiones de acceso simultáneo a una misma posición de memoria por parte de 2 procesos. Uno pide acceso en lectura y otro en escritura, si el hardware resuelve siempre el conflicto atendiendo primero al proceso que solicita la lectura, entonces el algoritmo de Dekker no sería equitativo.

2.3 Una solución simple al problema de la exclusión mutua: el algoritmo de Peterson

Peterson [Peterson, 1983] propuso una forma simple de resolver el problema de la exclusión mutua para dos procesos que, a partir de entonces, se considera la solución canónica a dicho problema. Esta solución permite obtener, además, una fácil generalización al caso de N-procesos, manteniendo la estructura del protocolo para el caso N=2.

2.3.1 Solución para 2 procesos

Las variables compartidas por los procesos son:

```
var
   c: array [0..1] of boolean;
   turno: 0..1;
```

El array `c` se inicializa a `FALSE` e indica si el proceso `Pi` intenta o no entrar en sección crítica. La variable `turno` sirve para resolver conflictos cuando ambos procesos intentan entrar simultáneamente.

Supuestos los valores `i=0` y `j=1` correspondientes a los identificadores de los procesos `Pi` y `Pj`. El código para el proceso `Pi` es el siguiente:

```
...
c[i]:= true;
turno:= i;
while(c[j] and turno=i) do
  nothing;
enddo;
 <seccion critica>
c[i]:= false;
...
```

Exclusión mutua
Para demostrar que el algoritmo anterior cumple la propiedad de exclusión mutua se sigue un razonamiento por reducción al absurdo. Suponer que `P0` y `P1` se encuentran ambos en sus secciones críticas, entonces los valores de sus claves han de ser necesariamente, `c[0]= c[1]= TRUE`; sin embargo, las condiciones de espera de los dos procesos no han podido ser simultáneamente ciertas, ya que la variable compartida `turno` ha debido tomar el valor final 0 ó 1 antes de que ambos procesos realicen iteraciones. Por tanto, sólo uno de los procesos ha podido entrar en sección crítica, digamos que es `Pi` ya que este encontró `turno=j`. `Pj` no pudo haber entrado en sección crítica junto con `Pi`, ya que para esto hubiera necesitado encontrar `turno=i`; sin embargo, la única asignación que el proceso `Pj` pudo haber hecho a la variable `turno` le era desfavorable.

Ausencia de interbloqueo
Suponer que `P0` está continuamente bloqueado esperando entrar en su sección crítica. Suponer también que el proceso `P1` está pasivo y no intenta ejecutarse, o bien que el proceso `P1` está esperando entrar en sección crítica continuamente. En el primer caso `c[1]= FALSE` y `P0` puede entrar en su sección crítica. El segundo caso es imposible, ya que la variable compartida `turno` ha de ser 0 ó 1, y hará a alguna de las condiciones de espera cierta, permitiendo entrar en sección crítica al proceso correspondiente. Por lo tanto, el suponer que ambos procesos están bloqueados indefinidamente lleva a contradicción.

Equidad de los procesos
Suponer ahora que `P0` está bloqueado esperando entrar en sección crítica y `P1` está entrando una y otra vez en ella, y por lo tanto monopolizando su acceso a la sección crítica. Sin embargo, esto también lleva a contradicción, ya que si `P1` intenta entrar de nuevo en sección crítica hará la asignación `turno=1` que validará la condición para que `P0` entre en sección crítica.

2.3.2 Solución para *N* procesos

El principio de generalización del algoritmo de Peterson es simple: utilizar repetitivamente (N-1)–veces la solución de 2 procesos para dejar esperando al menos 1 proceso cada vez hasta que sólo quede uno, el cual puede entrar en la sección crítica con seguridad de que se mantendrá la exclusión mutua mientras esté en ella.

Esquema general

Las variables compartidas necesarias son:

```
var c: array[0..N-1] of -1..N-2; /*los valores del array c representan
                                   las etapas por las que pasa un proceso
                                   antes de entrar en s.c.*/
   turno: array[0..N-2] of 0..N-1; /*representa el no. de proceso que tiene el
                                    turno en cada etapa, para resolver posibles
                                    conflictos*/
```

Los valores iniciales de todos los valores de los arrays `c` y `turno` son -1 y 0, respectivamente. `c[i]= -1` representa que `Pi` está pasivo y no ha intentado entrar en el protocolo. `c[i]= j` significa que `Pi` está en la etapa j-ésima del protocolo.

La implementación del algoritmo representa una cola con `N-1` posiciones para entrar en sección crítica. Los procesos bloqueados en una etapa pueden volver a ser adelantados por los demás, que salen de la sección crítica y vuelven a entrar al protocolo, en la siguiente etapa del protocolo.

El código siguiente pertenece al proceso `Pi`. Se utilizan las variables locales `i`, `j` y `k`. Los números de procesos se toman en el rango `i= 0..N-1`.

```
while true do
  begin
   Resto de las instrucciones;
   (1) for j=0 TO N-2 do
   (2)   begin
   (3)     c[i]:= j;
   (4)     turno[j]:= i;
   (5)     for k=0 TO N-1 do
   (6)       begin
   (7)         if (k=i) then continue;
   (8)         while(c[k]>=j and turno[j]=i) do
   (9)          nothing;
   (10)         enddo;
   (11)      end;
   (12)   end;
   (13) c[i]:= n-1; /*meta-instruccion*/
   (14) <seccion critica>
   (15) c[i]:= -1
  end
enddo;
```

Verificación formal de las propiedades del algoritmo

Se dice que el proceso `Pi` *precede* al proceso `Pk` sii el primer proceso está en una etapa más adelantada que el segundo, i.e. `c[i] > c[k]`.

La verificación de que el algoritmo satisface las propiedades que serían deseables (tanto seguridad como vivacidad) se realiza demostrando los cuatro lemas siguientes.

Lema 1
Un proceso que precede a todos los demás puede avanzar al menos una etapa.

Sea `Pi`, cuando dicho proceso evalúa su condición de espera encuentra que `j = c[i] > c[k]`, *para todo k distinto de i*, ya que precede a todos los demás. Por lo tanto, la condición para terminar la espera se hace cierta y puede avanzar a la etapa `j+1`.

Debido a que cuando un proceso comprueba su condición no realiza una instrucción atómica, puede ocurrir que durante dicha comprobación uno o varios procesos alcancen la misma etapa que `Pi`, pero tan pronto como el primero de dichos procesos modifique la variable `turno[j]`, asignándole su propio identificador, ocasionará que el proceso `Pi` pueda continuar (porque

dejaría de ser el último en asignar el turno de dicha etapa `j`). Después podría ocurrir que el proceso `Pi` fuera adelantado en la etapa siguiente, si hubiera llegado más de un proceso a la citada etapa `j` del algoritmo.

Lema 2
Cuando un proceso pasa de la etapa `j` *a la etapa* `j+1`*, se han de verificar alguna de las condiciones siguientes.*

(a) precede a todos los demás,
(b) o bien, no está solo en la etapa `j`

Suponer que `Pi` está a punto de avanzar a la etapa siguiente, entonces: o bien se da la condición del Lema 1, y por tanto también se verifica (a), o bien no se da dicha condición. En este último caso, para que el proceso `Pi` pudiera avanzar ha de cumplirse que `turno[j]` $\neq$ `i`, luego, en ese caso, el proceso `Pi` ha de estar acompañado por al menos otro proceso en la etapa `j` y además no ha sido el último proceso en asignar el turno de dicha etapa. Independientemente del número de procesos que modificaron `turno[j]` después de `Pi`, existirá un proceso `Pr` que fue el último en modificar `turno[j]`, dicho proceso no satisface la condición para salir de la espera, ya que `turno[j]= r`, y se queda bloqueado. El proceso `Pr` hace que la condición (b) del lema sea cierta.

También puede suceder que el proceso `Pi` vaya a avanzar de etapa porque ha encontrado que la condición (a) es cierta y mientras tanto otro proceso alcance su misma etapa, haciendo falsa dicha condición, pero en ese caso se haría cierta la condición (b).

Lema 3
Si existen al menos dos procesos en la etapa `j`*, entonces existe al menos un proceso en cada una de las etapas anteriores.*

La demostración se hace por inducción sobre la variable j.

Para (`j=0`) no tiene sentido dicha demostración. Para demostrar el caso (`j=1`) se utiliza el Lema 2. Si suponemos que existe un proceso en la etapa `j=1`, entonces otro proceso (o más de uno) se unirá a él, dejando atrás un proceso en la etapa `j=0`. Este proceso, que se ha quedado en la etapa 0, mientras que esté solo no puede avanzar.

Suponer ahora que el Lema 2 se satisface en la etapa `j-1`. Si hay 2 ó más procesos en la etapa `j`, entonces se ha de satisfacer que en el instante en que el último de dichos procesos abandonó la etapa `j-1` estaba ocupada (por el Lema 2) por, al menos, otro proceso. Por hipótesis de inducción todas las etapas anteriores han de estar ocupadas y por el Lema 2 ninguna de dichas etapas ha podido quedarse vacía desde entonces.

Lema 4
El número máximo de procesos que puede haber en la etapa `j` *es* `N-j`*, con* `0 <= j <= N-2`*.*

Aplicando el Lema 3, si las etapas `0..j-1` tienen al menos 1 proceso, entonces la etapa `j` tiene como máximo `N-j` procesos. Si alguna de las etapas anteriores estuviese vacía (caso de que un proceso se adelante a los demás y consiga avanzar siempre), entonces por el Lema 3 habrá como máximo un proceso en la etapa j. Por lo tanto, en la etapa `N-2` hay como máximo 2 procesos.

Exclusión mutua
Supongamos que hay 1 proceso en la etapa `N-2` y 1 proceso en la etapa `N-1` (sección crítica), entonces aplicando el Lema 2, el proceso de la etapa `N-2` no puede entrar en sección crítica ya que no precede a todos los demás y está solo en dicha etapa. Cuando el proceso que está en la etapa `N-1` abandone la sección crítica, se cumplirá la condición de que el proceso en la etapa `N-2` precede a los demás y entonces podrá avanzar.

Supongamos que hay 2 procesos en la etapa `N-2`, sólo uno de ellos podrá avanzar, ya que para el otro, cuando este avance, no se cumplirán las condiciones del Lema 2.

Ausencia de interbloqueo
La hipótesis de incorrección sería que los procesos se quedaran bloqueados al llegar a una etapa y no consiguiesen avanzar. Si suponemos que un proceso precede a todos los demás, entonces por el Lema 1 no puede quedarse bloqueado en ninguna etapa. Si ahora suponemos que un proceso llega a una etapa donde hay otros procesos, entonces los procesos que se encontraban en dicha etapa pueden avanzar.

Equidad
Supongamos que todos los procesos intentan entrar en sección crítica y que en la etapa 0 se encuentra el proceso `Pi` que ha sido el último en asignar `turno[0]` y por tanto se bloquea en dicha etapa. En el caso más desfavorable los restantes `N-1` procesos entran en sección crítica y vuelven al protocolo de entrada. Sea `Pl` el último de dichos procesos en asignar `turno[0]:= l`, luego dicho proceso hará cierta la condición de `Pi` para abandonar la espera. Como consecuencia `Pi` se desbloqueará y pasará a la etapa 1. La misma situación se aplicará a los `N-1` procesos restantes. Como consecuencia, el número de turnos que un proceso cualquiera tendría que esperar como máximo, en el caso más desfavorable de ejecución del algoritmo anterior para él, sería: $\mathrm{r(N)} = \mathrm{N} - 1 + \mathrm{r(N-1)} = \frac{\mathrm{N(N-1)}}{2}$ turnos.

2.4 Monitores como mecanismo de alto nivel

Los *monitores* derivan de una práctica antigua de programación de los sistemas operativos, denominada construcción de un *monitor monolítico*. Dichos monitores de los sistemas se ejecutan en un modo privilegiado, frente a otros programas del sistema o de los usuarios.

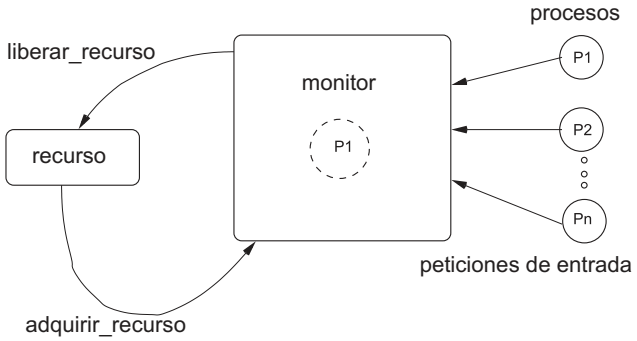


Figura 2.1: Representación gráfica de los elementos asociados a un modulo monitor

Cuando se ejecuta código de un monitor (se dice que el sistema *entra* en un monitor), se prohiben las interrupciones y, por tanto, se asegura la exclusión mutua en el acceso a los recursos protegidos. También es una práctica habitual en la programación de sistemas que, sólo cuando se ejecuta el código de un monitor, los programas tengan autorización para acceder a ciertas áreas de memoria o a ejecutar determinadas instrucciones de E/S.

Inspirándose en la práctica de programación anterior a nivel de sistema es como aparece la construcción denominada *Monitor* en los lenguajes de programación concurrentes. De esta manera, para garantizar la ejecución *segura* de un programa concurrente con múltiples procesos que acceden a recursos y a estructuras de datos comunes, se puede *centralizar* el acceso a dichos recursos críticos mediante la encapsulación de estos en una versión descentralizada[1] del *monitor monolítico*, que ahora pasará a poseer las características de un paquete o módulo.

El monitor atiende las peticiones de los procesos concurrentes de una-en-una, dejando los recursos que protege en un estado consistente antes de atender una nueva petición. En la figura 2.1 puede apreciarse que sólo se permite a un proceso (P_1, en el ejemplo) la ejecución de uno de los procedimientos del monitor. Se dice entonces que la *entrada* de un proceso en un monitor excluye la entrada de cualquier otro.

2.4.1 Definición y características de los monitores

Los monitores son módulos[2] de los programas concurrentes para *sistemas de computación centralizados*. De esta manera, un programa concurrente va a contener ahora dos tipos de *entidades*–software: los procesos, que son *entidades activas* de los programas; y los monitores, que tienen un carácter *pasivo*, ya que estos no inician ninguna computación, sino que ofrecen sus operaciones para que sean llamadas por los procesos, los cuales sólo pueden *interaccionar* invocando los procedimientos que proporcionan los monitores declarados en el programa.

[1] cada monitor estará encargado de una tarea específica y, por lo tanto, tendrá sus propios datos e instrucciones privilegiadas

[2] módulo es una *entidad* del software que agrupa procedimientos y datos relacionados

2.4.2 Centralización de recursos críticos

Se pueden diseñar distintos monitores para diferentes recursos dentro de un mismo programa concurrente, de esta forma el programa global resulta ser más eficiente, ya que la ejecución de monitores no relacionados puede ser realizada concurrentemente sin que se produzca *interferencia* en el acceso al conjunto de datos que protege cada uno. Además, se aumentaría el grado de robustez del programa, ya que un error o una excepción que pudiera levantarse durante el acceso una variable del monitor M1 no afectaría a las variables de otro monitor M2.

Estructuración del acceso a los datos del monitor

Un monitor es también un TDA (*Tipo de Datos Abstracto*)[3] cuyos procedimientos pueden ser ejecutados por turnos entre un conjunto de procesos concurrentes. La encapsulación de los datos previene que los procesos puedan acceder a la representación interna de estos y, de esta forma, se evita la interferencia en el acceso y, por tanto, que la ejecución concurrente de los procesos produzca valores inconsistentes en las variables que protege el monitor.

La sintaxis de los monitores incluye la parametrización del módulo monitor, lo cual permite declarar copias o *instancias* distintas de un mismo monitor dentro un programa concurrente. Cada una de dichas instancias representa una copia de las variables *permanentes* [4], de las estructuras de datos utilizadas en su implementación y de las señales utilizadas para la sincronización interna de los procedimientos.

Todo monitor define un *cuerpo* constituido por una secuencia de instrucciones (entre las instrucciones `begin` y `end`) que son ejecutadas automáticamente cuando se inicia el programa donde se declara el monitor. Dicho *cuerpo* se utiliza para proporcionar valores iniciales a las variables permanentes del monitor.

Las variables permanentes sólo son accesibles por los procedimientos del monitor y su ámbito y duración corresponden con los del propio monitor, a diferencia de los parámetros y de las variables locales de los procedimientos del monitor cuyo ámbito y duración coincide con el de estos.

La sintaxis correcta para llamar a un procedimiento del monitor suele incluir el nombre del monitor y el nombre del procedimiento que se desea ejecutar separados por un punto:

```
nombremonitor.nombreprocedimiento(...parametros actuales...);
```

Protección en el acceso a los datos del monitor

Para que las variables permanentes de un monitor no alcancen valores erróneos durante la ejecución, ya que son modificadas por parte de los procesos concurrentes del programa, no se puede utilizar dentro del texto de los procedimientos ninguna variable del programa concurrente

[3] un monitor = TDA + protección de los datos del monitor para acceso concurrente por los procesos del programa

[4] aquellas cuyos valores persisten después de ejecutarse una operación del monitor y antes de ejecutarse la siguiente operación que pudiera modificar dichos valores

```
Monitor nombre [( <parametros> )] -- para crear instancias del monitor
  var
    ... --declaracion de variables permanentes
  Procedure P1[( <parametros> )] -- los parametros de los
    begin                        -- procedimientos son la
      ...                        -- interfaz con los procesos
    end;                         -- de usuario
  ...
  Procedure Pn[( <parametros> )]
    begin
      ...
    end;
  begin -- cuerpo del monitor
    ... --inicializacion de las variables del monitor
  end;

<parametros>::= [var]<declaracion>{;[var]<declaracion>}
```

Figura 2.2: Notación sintáctica básica de los monitores

declarada fuera del monitor[5]. Como consecuencia de esto, la comunicación entre los procesos de un mismo programa concurrente ha de realizarse sólo a través de los parámetros de los procedimientos de sus monitores.

2.4.3 Operaciones de Sincronización y Señales de los Monitores

Los procesos de un programa concurrente con monitores no tienen por qué incluir en su código ninguna operación de sincronización, sólo tienen que llamar a los procedimientos del monitor, los cuales ya incluyen en su texto las operaciones de sincronización necesarias para la correcta interacción cooperativa entre dichos procesos, así como aseguran dejar los datos en un estado consistente al terminar su ejecución. De acuerdo con este modelo, los procedimientos del monitor pueden verse interrumpidos varias veces durante su ejecución, por lo que su código ha de poseer la propiedad de *reentrancia*[6].

Como se ha dicho anteriormente, la sincronización ha de ser explícitamente programada dentro de los procedimientos del monitor, utilizándose para ello un nuevo tipo de datos denominado *variables condición* o *señales*. Las *variables condición* se utilizan dentro de los procedimientos del monitor para poder separar aquellos procesos del programa que han de retrasar su ejecución hasta que el estado del monitor satisfaga una determinada condición, de ahí su nombre.

La condición por la que esperan los procesos viene definida a partir de los valores de las

[5]si no fuera así se podrían producir *condiciones de carrera* en el acceso, ocasionando que el valor final de la variable dependiera del entrelazamiento de las acciones elementales de los procesos

[6]código compartido por varios procesos que pueden ejecutar parte del mismo, interrumpirse y volver a ejecutarse donde se quedaron, sin que se produzca ninguna pérdida de información

```
Monitor recurso (r: integer);
  var ocupado: boolean; no_ocupado: cond;
  procedure adquirir_recurso;
    begin
      if ocupado then no_ocupado.wait();
      ocupado:= true;
    end;

                                        procedure liberar_recurso;
                                          begin
                                           ocupado:= false;
                                          no_ocupado.signal();
                                          end;
                                        begin
                                          ocupado:= false;
                                        end;
```

Figura 2.3: Operaciones de acceso a recurso programadas como un monitor simple.

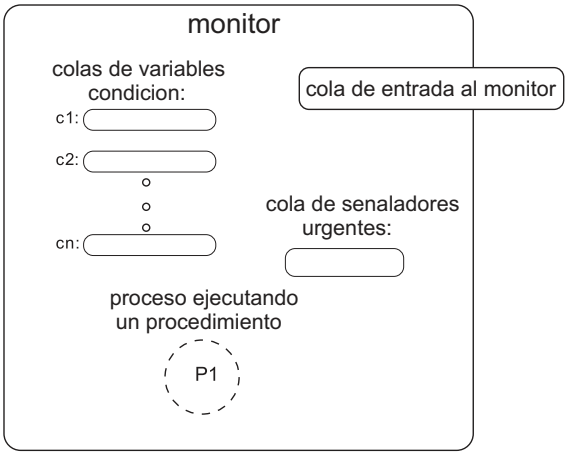


Figura 2.4: Representación gráfica de las colas en la implementación de los monitores.

variables permanentes del monitor. Se tiene una operación de sincronización para bloquear a los procesos hasta que se cumpla dicha condición y otra para reanudar la ejecución de un proceso bloqueado cuando ésta se convierte en verdadera.

La declaración de las *variables condición* se realiza en la sección de declaración de variables permanentes del monitor, como objetos de tipo `cond`, no siendo necesario el inicializarlas, ya que no está previsto que posean ningún valor asociado.

En el ejemplo anterior, ver figura 2.3, los procesos han de esperar sólo si el único recurso que protege el monitor está ocupado. Se necesita, por tanto, una sola variable condición para sincronizar correctamente a los procesos que llaman a sus procedimientos. Así pues, gracias a las variables condición, se puede tener más de 1 proceso *esperando reanudarse* dentro de la cola de condición correspondiente, pero sólo un proceso activo dentro del monitor,ver figura 2.4.

El estado de espera de los procesos se representa internamente en los monitores, bloqueándolos en una cola hasta que la condición se haga cierta. La representación interna de estas colas, ver figura 2.4, no es accesible a los procesos de la aplicación que utilizan el monitor[7]. Para cada condición lógica que pueda implicar la espera de los procesos del programa, el programador

[7]el programador que utilice un monitor en su programa no puede inspeccionar el contenido de `c`, ni ordenar desbloquear selectivamente a un proceso bloqueado de la cola, etc.

debería asociarle una variable condición[8], declarada dentro del monitor correspondiente.

Para bloquear/desbloquear a los procesos en las colas de las variables condición se incluyen un par de operaciones denominadas `wait()` y `signal()`, similares a las de los semáforos, aunque con una semántica totalmente diferente. Las *variables condición*, a diferencia de los *semáforos*, no guardan ningún valor interno protegido que indique el número de veces que se permite ejecutar la operación de espera (`wait()`) sin bloquear:

- `c.wait()`: el proceso que llama a esta operación se bloquea siempre y entra en la cola de procesos bloqueados asociada a la variable condición `c`, esperando que la condición sea cierta. Se planifica el desbloqueo de los procesos según un orden FIFO.

- `c.signal()`: si la cola de la variable condición `c` no está vacía, entonces desbloquea al primer proceso esperando en dicha cola. Si no hay ningún proceso bloqueado en la cola es una operación nula, no tiene efecto.

Al ejecutarse la operación `c.wait()`, el monitor ha de quedar libre para que pueda entrar otro proceso al monitor, ya que si no fuera así los procedimientos del monitor quedarían inaccesibles para los otros procesos y, posiblemente, el programa completo terminaría bloqueándose.

2.4.4 Semántica de las señales

La definición clásica de las señales de los monitores supone que el proceso que envía la señal (señalador) ha de ceder el acceso al monitor[9] al proceso señalado (desbloqueado en la cola de la condición), de esta forma se impide que otros procesos esperando en la cola de entrada al monitor se adelanten y pospongan la reanudación del proceso señalado. Dicho efecto se le conoce con el nombre de *robo de señal* y ha de ser evitado en una buena práctica de programación con monitores, ya que el proceso *intruso* podría cambiar el valor lógico de una condición de desbloqueo, posiblemente ocasionando que las variables permanentes alcanzasen valores inconsistentes cuando posteriormente el proceso señalado consiguiera entrar en el monitor. Por consiguiente, las señales de *semántica desplazante* suponen la *reanudación inmediata* del proceso señalado y evitan que pueda sufrir un robo de señal por parte de un proceso intruso.

Pero la semántica desplazante no es la única para implementar la sincronización en los monitores. Por ejemplo, un proceso al ejecutar la operación `c.signal()` podría pensarse que no abandone inmediatamente el monitor, quizás el proceso señalado podría bloquearse temporalmente hasta que el señalador termine y abandone el monitor. La semántica del tipo de señales que implemente un lenguaje con monitores afectará principalmente al comportamiento concreto que seguirá el proceso que señala y ha de asegurar que, en cualquier caso, sólo pueda existir, como máximo, un proceso dentro del monitor durante toda la ejecución del programa.

Existen cinco tipos de mecanismos que son utilizados por diferentes lenguajes concurrentes con monitores para enviar señales a los procesos bloqueados en una cola de condición. Cada uno de estos mecanismos de señalación posee una semántica diferente:

[8] programándola explícitamente, o bien, de manera implícita, comprobando que se satisface antes de programar la operación `signal` correspondiente

[9] a esta semántica de las señales se le conoce con el nombre de *desplazante*

SA	señales automáticas	señal implícita
SC	señalar y continuar	señal explícita, no desplazante
SX	señalar y salir	señal explícita, desplazante, el proceso sale del monitor
SW	señalar y esperar	señal explícita, desplazante, el proceso señalador espera en la cola de entrada al monitor
SU	señales urgentes	señal explícita, desplazante, el proceso señalador espera en la cola de *procesos urgentes*

Tabla 2.1: Diferentes mecanismos de señalación en monitores.

- SA: es un mecanismo para enviar señales de forma implícita. Esto es, las señales son incluidas por el propio compilador cuando se genera el código del programa. El programador sólo tiene que incluir las sentencias `c.wait()` en el texto de los procedimientos del monitor, donde sea necesario. El sistema en tiempo de ejecución se encargará de reanudar la ejecución de procesos bloqueados cuando el programa alcance un estado en el cual la condición por la que esperan se convierta en cierta.

- Por el contrario, SC, SX, SW y SU, son señales explícitas. El programador ha de incluir suficientes operaciones `c.signal()` cuando escriba el texto de los procedimientos del monitor. Si no lo hiciera así, entonces los procesos que usan el monitor podrían bloquearse indefinidamente.

- SA, SC: son señales con semántica no *desplazante*. El proceso que ejecuta la operación `c.signal()` no deja libre el monitor inmediatamente, sino que se sigue ejecutando hasta que termine la ejecución del procedimiento, o se bloquee porque ejecute posteriormente una operación `c.wait()`.

- SX, SW, SU: son desplazantes. Un proceso que ejecuta la operación `c.signal()` es obligado a ceder el procesador al primer proceso bloqueado en la cola de condición `c`, si dicho proceso existiera, si no, seguiría ejecutándose y la operación no tendría efecto en el estado del monitor. La condición asociada al envío de la señal debe ser cierta cuando un proceso ejecuta esta operación y, gracias a la semántica desplazante de estas señales, sigue siendo cierta cuando el proceso señalado reanude su ejecución. Nótese que si suponemos una semántica diferente, como sucede en el caso de las SC, no se puede suponer esto, ya que otro proceso, que pudiera entrar en el monitor antes que el proceso señalado, podría haber cambiado el valor de la condición.

Por otra parte, los tres tipos de señales siguientes provocan un comportamiento diferente del proceso señalador después de ejecutar la operación `c.signal()`:

- SX: se fuerza al proceso a salir del monitor, por tanto con este tipo de semántica de señal hay que programar siempre la operación `c.signal()` como la última instrucción del procedimiento del monitor donde aparezca.

- SW: el proceso que envía se bloquea en la cola de entrada al monitor.

- SU: se hace esperar al proceso que envía la señal en una cola de procesos *urgentes*, ver figura 2.4. Dichos procesos son prioritarios, para reanudarse y entrar de nuevo en el monitor con respecto a otros procesos que pudieran estar intentado la entrada al mismo[10].

[10]tales procesos estarían bloqueados en la cola de entrada al monitor

2.4.5 Buenas prácticas en la programación con señales de los monitores

En nuestros programas hemos de asegurarnos que los procesos se bloquean sólo cuando sea necesario. Es responsabilidad del programador el incluir suficientes operaciones `c.signal()` para el conjunto de procesos bloqueados en la cola de `c`, en el texto de los procedimientos del monitor para desbloquearlos cuando el programa alcance un estado en el cual se cumpla la condición contraria a la de bloqueo asociada a `c`. Si permaneciesen algunos procesos bloqueados indefinidamente en alguna cola de variable condición, entonces el programa no podría ser considerado totalmente correcto, ya que no se cumpliría la propiedad de *vivacidad*.

Un programa que utilice monitores se ejecutará más eficientemente si la operación `c.signal()` se ejecuta sólo cuando sea necesario. La ejecución de la operación `c.signal()` ocasiona un cambio de contexto[11], por tanto, debe ejecutarse sólo si hay procesos bloqueados esperando que se haga cierta la condición. Para evitar cambios de contexto innecesarios, se debe utilizar la operación `c.queue()` que devuelve el valor *true* si hay procesos bloqueados esperando la condición y *false* si no los hay.

Algunos lenguajes de programación con monitores también incluyen `c.signal_all()`, que ocasiona el desbloqueo de todos los procesos que se encuentren en la cola de la variable condición `c`. El efecto de esta operación es el mismo que si se ejecutase:

```
while c.queue() do c.signal() end enddo;
```

El bucle anterior nunca debe programarse dentro de los procedimientos si se utilizan monitores con señales de semántica desplazante, ya que la ejecución de dicho bucle ocasionaría la salida de todos los procesos en una cola y se produciría una condición de carrera al entrar de nuevo aquellos que no cumplan la condición de desbloqueo.

2.4.6 Variables condición con prioridad

Para determinadas clases de aplicaciones podría existir la necesidad de utilizar variables condición que planifiquen el desbloqueo de los procesos según un orden prioritario, en lugar de respetarse el orden en el cual los procesos se bloquearon[12], como se supone en la definición estándar de las operaciones de sincronización de los monitores. Para poder tener un orden de desbloqueo de los procesos según su prioridad se introduce una nueva operación definida para las variables condición que se denomina *wait prioritario* y cuya definición es la siguiente:

- `c.wait(prioridad)`: bloquea los procesos en la cola `c`, pero ordenándolos de forma automática con respecto al valor que tenga el argumento `prioridad` cuando se llamó a la operación.

- `prioridad`: número no negativo que indica mayor importancia del proceso para valores numéricos más pequeños.

[11] ya que el proceso ha de interrumpir momentáneamente su ejecución para que el ejecutivo del lenguaje de programación compruebe si hay procesos bloqueados en la cola de `c`

[12] y como consecuencia de ello entraron en la cola FIFO de una variable condición *ordinaria* (no-prioritaria)

A continuación se presenta un ejemplo de monitor para un programa de un sencillo *despertador* que permite disparar una alarma a diferentes horas, dependiendo de las peticiones que hayan hecho sus procesos-usuarios. El monitor puede *recordar* los tiempos que solicitaron sus usuarios y atender cada una de las peticiones a la hora indicada. Se ha programado creando un proceso[13] por cada una de las peticiones. Cuando se cumple el tiempo, el monitor envía una señal al proceso correspondiente, o a más de uno si varios de ellos indicaron la misma hora en su petición, para que se inicie una actividad de la aplicación, comience a sonar un timbre, se escriba en un archivo de registro, etc.

Los procesos pasan a bloquearse en la cola de la variable condición prioritaria `despertar`, después de realizar la llamada al procedidimiento `despiertame()`. Se supone que el procedimiento `tick()` está *conectado* con la interrupción de reloj del computador en el que se vaya a programar la aplicación que utilice el siguiente monitor:

```
Monitor despertador;
  var
    ahora: integer;
    despertar: cond; --prioritaria
  Procedure despiertame(n: integer);
    var alarma: integer;
    begin
      alarma:= ahora + n;
      while ahora<alarma do
        despertar.wait(n);
      end do;
      despertar.signal(); -- el proceso que se despierta, despierta al siguiente
    end;              -- y asi hasta el 'ultimo
  Procedure tick();
    begin
      ahora:= ahora+1;
      despertar.signal();
    end;
 begin
   ahora:= 0;
 end;
```

Figura 2.5: Monitor que implementa una alarma programado con una señal prioritaria.

Las variables condición con prioridad se pueden simular con variables condición FIFO, pero siempre resultará menos eficiente que una implementación interna llevada a cabo por el propio lenguaje de programación.

[13] mejor una *hebra* si el lenguaje de programación lo permite

2.5 Lenguajes de Programación con Monitores

El desarrollo del concepto de programación que respresentan los monitores tiene su inmediato antecedente en el concepto de *clase*, que apareció por primera vez en un lenguaje llamado SIMULA-67 propuesto, principalmente, por Ole-Johan Dahl [Dahl et al., 1970]. La idea de asociar la encapsulación de datos con la exclusión mutua en el acceso a un recurso compartido por los procesos, que es el fundamento del concepto monitor, se debió conjuntamente a Edsger W. Dijkstra [Dijkstra, 1971], Per Brinch-Hansen [Hansen, 1977] y C.A.R. Hoare [Hoare, 1999]. Como consecuencia de las ideas de estos investigadores, los monitores fueron incluidos en una colección inicial de lenguajes de programación concurrentes, la mayoría de ellos fueron propuestos durante los años 70. El lenguaje de programación concurrente denominado Concurrent Pascal [Brinch-Hansen, 1975] fue el primero en ofrecer los monitores como una construcción sintáctica para la programación de sistemas. Lenguajes concurrentes posteriores con monitores han sido: Modula [Wirth, 1985], Mesa [Lampson and Redell, 1980], Pascal Plus [Welsh and Bustard, 1979], Concurrent Euclid [Holt, 1983] y Turing Plus [Holt et al., 1987]. El problema de la anidación de llamadas fue identificado por Andrew Lister [Lister, 1977].

El lenguaje de programación Java ha replanteado nuestra comprensión acerca de los monitores incluyendo dicho concepto en un lenguaje de programación orientado a objetos. Java puede ser considerado como el lenguaje de programación más difundido que ha utilizado el concepto de monitor para desarrollar programas concurrentes con hebras.

2.5.1 Programación Concurrente en Java

El lenguaje de programación Java contiene una serie de construcciones y clases específicas para poder realizar programación concurrente. Entre las más importantes podemos citar:

- `java.lang.Thread`, utilizada para el inicio y control de hebras.
- `java.lang.Object`, que incluye la instrucción de espera y las *notificaciones*:
  1. `wait()`
  2. `notify()`
  3. `notifyAll()`
- Los `cualificadores` necesarios para proteger código y definir la duración de su ejecución:
  - `synchronized`,
  - `volatile`

Programar con elementos concurrentes en las aplicaciones, con la creación de hebras (o *threads*) en los programas, es algo fundamental para conseguir la implementación de buen código en Java, ya que permite obtener:

- Animaciones de *applets*.
- Programación de servidores de red eficientes.

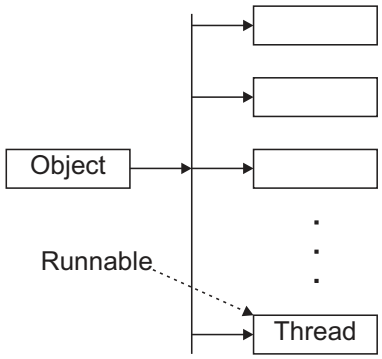


Figura 2.6: Situación de `Thread` dentro de la jerarquía de clases en Java.

2.5.2 Creación de hebras

El entorno de programación Java, dentro de las aplicaciones, impone la obligación de implementar la interfaz `Runnable` y, por tanto, la de llamar `run()` al método principal programado dentro de una clase que deseemos convertir en una hebra. Existen varias alternativas para crear clases-hebra en Java, que repasamos a continuación.

- Extender la clase Thread y redefinir el método público run (ver figura 2.7):

```
class A extends Thread{
 public A(String nombre) { super(nombre);}
 public void run(){
  System.out.println("nombre= " + getName());
 }
}
class B{
 public static void main(String[] args){
  A a= new A("Mi hebra");
  a.start();
 }
}
```

Figura 2.7: Creación de una clase hebra como extensión de `Thread`

- Implementar la interfaz Runnable en una clase con método público run() (ver figura 2.8):

La segunda forma es mejor, ya que en Java no existe la herencia simultánea de varias clases (herencia múltiple). Si para crear una nueva clase–hebra extendemos a la clase `Thread`, la primera perdería la posibilidad de heredar de cualquier otra clase. Por otra parte, desde un punto de vista práctico, la creación del objeto hebra se puede incluir dentro del método constructor de la clase, de esta forma conseguimos una mejor encapsulación , ya que el constructor reflejaría el hecho de que al llamarlo también se crea una hebra que soporta a cada nueva instancia de dicha clase (ver figura 2.9).

Además, la clase Thread posee 4 constructores, que aceptan como argumento cualquier objeto (o grupo) que sea *conforme* con la interfaz `Runnable`, devolviendo una referencia a un objeto de la clase `Thread` o `ThreadGroup`:

Crear un clase que implemente la interfaz `Runnable`

```
class A extends OtraClase implements Runnable{
  public void run(){
    System.out.println("nombre= " + Thread.currentThread.getName());
  }
}
```

Crear un instancia de la clase anterior y pasarla al constructor de `Thread`

```
class B{
 public static void main(String[] args){
   A a= new A();
   Thread t= new Thread(a, "A");
   t.start();
 }
}
```

Figura 2.8: Creación de una clase hebra en dos pasos

```
class A extends OtraClase implements Runnable{
  private Thread t = null;
  public A(){
    t = new Thread(this, "A");
    t.start();
  }
  public void run(){
    ... }
}
```

Figura 2.9: Creación de la nueva hebra dentro del constructor de la clase

- `public Thread(Runnable objeto)`
- `public Thread (Runnable objeto, String nombre)`
- `public Thread (ThreadGroup grupo, Runnable objeto)`
- `public Thread(ThreadGroup grupo, Runnable objeto, String nombre)`

2.5.3 Estados de una hebra

Cuando se crea el objeto hebra `Thread t = new Thread(a);` en un programa, a partir de una referencia `'a'` de un objeto de una clase que implemente la interfaz `Runnable`, lo que obtenemos es un *esqueleto* de hebra, potencialmente ejecutable, y que estaría en un estado pasivo inicial, que denominados hebra *nueva* (ver figura 2.10).

Posteriormente, cuando se llama al método `t.start();`, una referencia (*thread identificator*) pasará a ser visible por el planificador de hebras del sistema; decimos que la nueva hebra ha pasado al estado *ejecutable*. Cuando el método `run()` de la hebra correspondiente finalmente pase a ejecutarse en algún procesador diremos que alcanza el estado *ejecutando*.

Además de los métodos de creación de hebras, anteriormente introducidos, la clase `Thread` admite la ejecución de una serie de *métodos de clase*; entre los más utilizados se encuentran los siguientes:

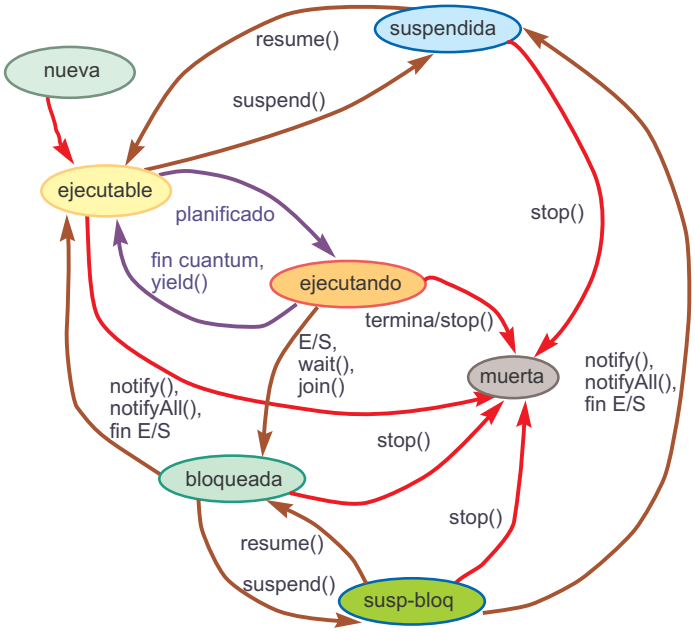


Figura 2.10: Estados de una hebra.

- `Thread.sleep(int); \\ tiempo en milisegundos`
- `Thread.yield(); \\cede el procesador`

Si una hebra ejecutándose en algún procesador ejecuta el metodo `yield()`, entonces abandona el procesador y pasa al estado *ejecutable*. Una hebra pasará al estado *bloqueada* cuando llama al método `sleep()`, volviendo al estado ejecutable cuando retorne la llamada de dicho método. La utilización correcta es `Thread.sleep(tiempo_milisegundos)`.

La clase `Thread` admite los siguientes *métodos de instancia* más comunes:

- start()
- join()
- stop()
- suspend(), resume()
- setPriority(int), getPriority()
- setDaemon(true)
- getName()

Una hebra en estado *ejecutable* o *ejecutando* pasa al estado suspendida cuando se invoca su método `t.suspend();`. Este método puede ser llamado por la misma o por otra hebra. La hebra volverá al estado *ejecutable* cuando su método `resume()` sea invocado por otra hebra. Las hebras pasan al estado *bloqueada* cuando llaman a `wait()` dentro de un bloque o de un método sincronizado, volviendo al estado ejecutable cuando otra hebra llame a `notify()` o a

`notifyAll()`. También se bloquean cuando llaman al método `join()` para sincronizarse con la terminación de otra hebra que aún no haya terminado su ejecución. Asímismo, cuando ejecutan alguna operación de E/S que implique una transferencia de datos bloqueante, como una lectura síncrona de un disco, etc. Una hebra se encuentra en el estado *bloqueada-suspendida* cuando la hebra, estando bloqueada, resulta además suspendida por otra hebra. Cuando la operación bloqueante termine, la hebra pasará al estado suspendido. Si, estando aún en el estado anterior, otra hebra invoca su método `resume()`, la hebra vuelve al estado *bloqueada*. Cuando termina la ejecución del método `run()` de una hebra o si se invoca su método `stop()`, entonces la hebra pasará al estado *muerta*.

2.5.4 Monitores en Java

Los monitores, tal como aparecen definidos en el artículo clásico de C.A.R. Hoare [Hoare, 1999], difieren en una serie de importantes aspectos con respecto a cómo se programa para conseguir una construcción similar en Java. Las variables condición, y sus colas de espera asociadas, se declaran explícitamente en la propuesta original, pudiéndose declarar más de una variable condición en el mismo monitor. La programación con *monitores de Hoare* contrasta con la de los monitores en Java, ya que este lenguaje de programación sólo permite una única cola de condición implícita en el programa donde se bloquean las hebras que ejecutan la operación `wait()`. En lo que sigue nos centraremos en la programación de los monitores de Java. En los programas con monitores, si se separan conjuntos de hebras y se les hace esperar en colas de variables condición diferentes, se propicia menos expulsión y reactivación de hebras. En Java, todas las hebras bloqueadas deben ser *despertadas* para volver a comprobar si se cumplen las condiciones por las que esperaban. Si después de ser despertada no se cumpliera la condición de desbloqueo para una hebra, ha de ser bloqueada de nuevo. Por otra parte, en la programación práctica con Java, las hebras que esperan a condiciones diferentes, normalmente lo hacen en momentos distintos de la ejecución de la aplicación y, por tanto, se elimina el coste extra debido a la reactivación de hebras y su bloqueo subsiguiente. Incluso, cuando esto último se diera, el coste de la replanificación extra de hebras no suele ocasionar mayor problema.

Código y métodos sincronizados

El cualificador `synchronized` sirve para hacer que un bloque de código o un método sea protegido por el cerrojo interno de los objetos, es decir, que impida el acceso al mismo por parte de más de una hebra del programa concurrentemente. Por tanto, las hebras tienen que adquirir el cerrojo de un objeto (`obj`) de Java previamente a ejecutar cualquier código sincronizado:

```
synchronized (obj) {
   bloque de codigo sincronizado
}
```

Si todas las secciones críticas a las que se necesite acceder se encuentran en el código de un único objeto, podremos utilizar `this` para referenciarlo:

```
synchronized (this) {
   bloque de codigo sincronizado
}
```

El cuerpo entero de un método podría ser código sincronizado:

```
tipo metodo ( ... ) {
   synchronized (this) {
      codigo del metodo sincronizado
   }
}
```

La siguiente construcción sería equivalente a la anterior:

```
synchronized tipo metodo ( ... ) {
   codigo del metodo sincronizado
}
```

Por tanto, la declaración de una clase con todos sus métodos, susceptibles de ser ejecutados por hebras concurrentemente, sincronizados es la forma de crear un monitor en Java.

```
class Contador { // monitor contador
  private int actual;
  public Contador (int inicial) {
     actual = inicial;
  }
  public synchronized void inc (){
     actual++;
  }
  public synchronized void dec (){
     actual--;
  }
  public synchronized int valor (){
     return actual;
  }
}
```

```
class Usuario extends Thread {
   private Contador cnt;
   public Usuario (String nombre,
                   Contador cnt) {
      super (nombre);
      this.cnt = cnt;
   }
   public void run () {
    for (int i=0; i<1000; i++) {
      cnt.inc ();
      System.out.println ("Hola, soy " +
          this.getName() +
          ", mi contador vale" +
          cnt.valor());
    }
   }
}
```

Figura 2.11: Programa con monitor Java para un contador *thread-safe*

2.5.5 Operaciones de sincronización y notificaciones

Otra diferencia importante de los *"monitores"* de Java respecto de la definición clásica de estos es la correspondiente a la semántica de la señalación a hebras que esperan. Las señales se denominan *notificaciones* en Java y la operación `notify()` posee una semántica similar a la de las señales SC. En Java la hebra notificada no tiene prioridad alguna para entrar inmediatamente al monitor, por tanto, pasará del estado *bloqueada* (en la cola de espera) al *ejecutable*, uniéndose a la cola de hebras *preparadas* del planificador del sistema en tiempo de ejecución. Además, la hebra que invoca la operación `notify()` no está obligada a dejar libre el monitor y puede continuar ejecutándose. Las condiciones por las que esperan las hebras han de ser siempre vueltas a comprobar en Java antes de que éstas puedan adquirir el cerrojo del monitor, ya que la condición puede haber sido invalidada en el lapso de tiempo desde que fueron notificadas.

Las señales con semántica desplazante de los monitores clásicos poseen, por tanto, la ventaja de que las condiciones de espera no tienen que ser comprobadas nuevamente, ya que la hebra señalada tiene prioridad para adquirir el cerrojo del monitor, respecto de nuevas hebras que quieran entrar al mismo. Como desventajas, podemos decir que las señales desplazantes son más complejas de implementar en un lenguaje de programación que las notificaciones previstas en Java. Podría significar pagar un precio derivado de la pérdida de eficiencia, que en determinadas aplicaciones no estaría justificado. `notify(), notifyAll(), wait()` sólo se pueden utilizar dentro de métodos (`synchronized`) que mantengan bloqueado el cerrojo asociado al objeto.

2.5.6 API Java 5.0 para programación concurrente

Hasta ahora nos hemos centrado en las APIs de bajo nivel que han formado parte de la plataforma Java desde sus primeras versiones. Tales primitivas son adecuadas para realizar tareas muy básicas, pero necesitaremos unos *bloques de construcción* de más alto nivel de abstracción para poder llevar a cabo tareas más avanzadas. Esto es cierto sobre todo cuando se preteden desarrollar aplicaciones masivamente concurrentes que exploten completamente las posibilidades de los sistemas multiprocesador y multi-core actuales.

En esta subsección estudiaremos algunas de las primitivas concurrentes de alto nivel que fueron introducidas en la versión 5.0 de la plataforma Java. La mayoría de estas primitivas han sido implementadas en los nuevos paquetes `java.util.concurrent`. También se encuentran nuevas estructuras de datos concurrentes en la infraestructura *Java Collections*.

Entre las primitivas de la API de alto nivel de Java, despiertan más interés entre los programadores:

- Objetos *lock* que soportan instrucciones de bloqueo que simplifican la programación de muchas aplicaciones concurrentes.
- Ejecutores que definen una API de alto nivel para lanzar y gestionar las hebras. Las implementaciones del tipo *Executor* que proporciona `java.util.concurrent` ofrecen la gestión de *pools de hebras* adecuados para programación de aplicaciones a gran escala.
- Las *colecciones concurrentes* hacen mucho más sencillo el gestionar grandes colecciones de datos, y pueden reducir en mucho la necesidad de sincronización entre las hebras de las aplicaciones.
- Las variables atómicas que poseen características para minimizar la sincronización y ayudan a evitar errores de consistencia de memoria.
- `ThreadLocalRandom` (presente en el JDK 7) proporciona una generación eficiente de números pseudo-aleatorios en múltiples hebras.

Cerrojos y variables condición

El método `lock()` deja la hebra que lo llama esperando de forma ininterrumpible hasta que el cerrojo se quede libre. `LockInterruptibly()`, se comporta de una manera análoga a la operación `lock()`, excepto que la espera podría ser interrumpida por otra hebra o por el sistema. El método `Condition()` crea una nueva *variable condición* que ha de ser utilizada

junto con un objeto `Lock` asociado. El método `TryLock()`, que después de ser llamado por una hebra devuelve el valor `true` si el `lock` se encuentra disponible. Este método admite como argumentos un plazo de tiempo (*timeout*) y la unidad en la que se mide dicha espera. Devuelve el valor `true` si el `lock` se convierte en disponible durante dicho *timeout*. El método `unlock()` convierte en disponible al cerrojo, para que pueda ser adquirido por otra hebra.

```
public interface Lock {

 public void lock();

 public void lockInterruptibly()
   throws InterruptedException;

 public Condition newCondition();

 public boolean tryLock();


 public boolean tryLock(long time,
                        TimeUnit unit)
        throws InterruptedException;


 public void unlock();
 }
```

```
package java.util.concurrent.locks;
public interface Condition {
  public void await()throws
              InterruptedException;
  public boolean await(long time,
                       TimeUnit unit)
        throws InterruptedException;
  public long awaitNanos(long
                         nanosTimeout)
        throws InterruptedException;
  public void awaitUninterruptible();
      // As for await, but not
      // interruptible.
  public boolean awaitUntil(
          java.util.Date deadl)
  throws InterruptedException;
      // As for await() but with
      // a timeout
  public void signal();
      // Wake up one waiting thread.
  public void signalAll();
      // Wake up all waiting threads.
}
```

Figura 2.12: Interfaces de Java 5.0 para cerrojos y variables condición

Las variables `Condition`[14] proporcionan las operaciones `await()` y sus variantes para suspender la ejecución de una hebra, hasta que ésta sea notificada por otra hebra de que alguna condición de estado podría ser actualmente cierta. Dado que el acceso a esta información de estado compartida ocurre en hebras diferentes, debe ser protegida de tal forma que algún tipo de cerrojo ha de estar permanentemente asociado a la evaluación y notificación de la condición.

La característica clave asociada a la operación de esperar a una condición consiste en que la llamada a dicha operación libera de forma ininterrumpible el cerrojo asociado a la variable condición y suspende a la hebra que llama, de una forma totalmente análoga a la ejecución de `Object.wait()`. Por ejemplo, supongamos que tenemos un *buffer* limitado cuyo contenido es modificado llamando a los métodos `put()` y `take()` (ver figura 2.13). Si el método `take()` se intenta llamar cuando el buffer está vacío, entonces la hebra se bloqueará hasta que el buffer contenga algún elemento; si se intenta el método `put()` cuando el buffer está lleno, entonces la hebra se bloqueará hasta que llegue a haber espacio libre en el buffer. Nos gustaría conseguir que las hebras que esperan ejecutar la operación `take()` y aquellas que esperan a la operación `put()` fueran situadas en colas de espera diferentes, de tal forma que podamos notificar de forma separada a una hebra que espera insertar o eliminar cada vez que aparezcan huecos

[14] objetos conformes con la interfaz `Condition` que se obtienen llamando al método `newCondition()`

```
class BoundedBuffer {
   final Lock lock = new ReentrantLock();
   final Condition notFull  = lock.newCondition();
   final Condition notEmpty = lock.newCondition();

   final Object[] items = new Object[100];
   int putptr, takeptr, count;

   public void put(Object x) throws InterruptedException {
     lock.lock();
     try {
       while (count == items.length)
         notFull.await();
       items[putptr] = x;
       if (++putptr == items.length) putptr = 0;
       ++count;
       notEmpty.signal();
     } finally {
       lock.unlock();
     }
   }
   public Object take() throws InterruptedException {
     lock.lock();
     try {
       while (count == 0)
         notEmpty.await();
       Object x = items[takeptr];
       if (++takeptr == items.length) takeptr = 0;
       --count;
       notFull.signal();
       return x;
     } finally {
       lock.unlock();
     }
   }
 }
```

Figura 2.13: Monitor Buffer Limitado implementado con variables `condition`

o elementos en el buffer, respectivamente. Esto se puede conseguir creando dos instancias distintas de `Condition`, tal como se puede ver en el ejemplo.

Los métodos de las variables condición de la figura 2.13 incluyen a `await()`, que libera el cerrojo asociado al monitor de forma atómica y ocasiona que la hebra actual espere hasta que: (a) otra hebra llame al método `signal()` y la primera sea escogida como la hebra que va a ser *despertada*; (b) otra hebra llame al método `signalAll()`, ocasionando que todas las hebras que esperan sean despertadas; (c) otra hebra interrumpa la espera de la primera hebra; o bien, (d) se despierte la hebra de forma espúrea o imprevista. Cuando el método `await()` vuelva está garantizado que la hebra que lo llamó poseerá el cerrojo (o `lock`) asociado a dicha variable condición.

2.6 Implementación de los Monitores

2.6.1 Implementación de las señales SX

Las señales SX tienen la ventaja de que pueden ser implementadas directamente en un lenguaje de programación concurrente, sin necesidad de utilizar otras primitivas de sincronización de más *bajo nivel* del sistema, como serían, por ejemplo, los semáforos. Sin embargo, el estudiar una posible implementación con semáforos del mecanismo de señales SX para monitores, aunque no sería necesaria, ayuda a entender mejor la semántica de este tipo de señales. Una posible implementación con semáforos sería la que aparece en la figura 2.14.

```
entrada:
  wait(s);

c.wait():
  cont_cond:= cont_cond + 1; --Variable entera que cuenta el numero
  signal(s);                 -- de procesos esperando.
  wait(sem_cond);            -- Se bloquea esperando. La cola del semaforo
  cont_cond:= cont_cond -1;  -- simula la cola de condicion.

c.signal():
  if cont_cond > 0 then signal(sem_cond) --reanudacion inmediata del primer
                   else signal(s)        --proceso bloqueado en la cola FIFO.
```

Figura 2.14: Implementación de las señales con semántica SX con semáforos

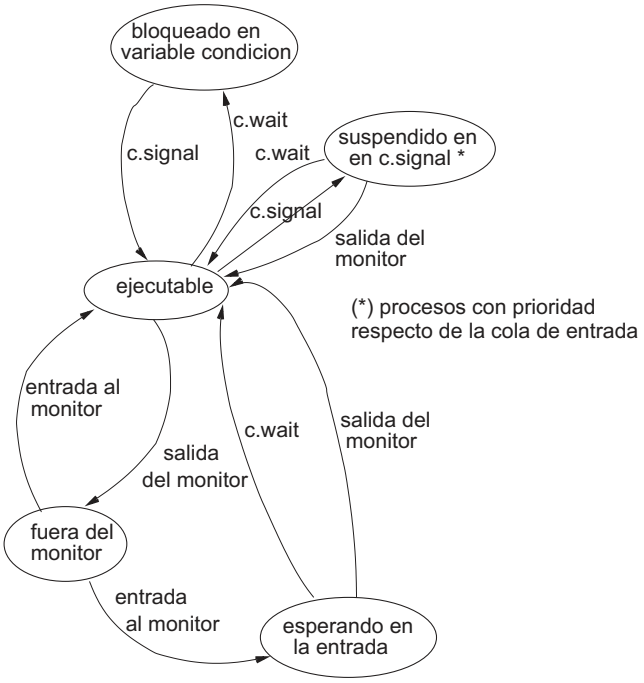


Figura 2.15: Estados de los procesos que usan un monitor con semántica de señales SU

Debido a que, según la semántica de este tipo de señales, hemos restringido la aparición de la operación `c.signal()` a ser la última instrucción de los procedimientos del monitor, el

desbloqueo de un proceso que espera la señal `semcond` puede ser combinado, en una misma instrucción (`if cont_cond>0 ...`), con dejar libre la entrada al monitor (`signal(s)`) a otros procesos. Por tanto, si hay procesos esperando una señal, es decir se cumple que `cont_cond>0`, entonces se señala a estos con prioridad sobre los que esperan entrar en el monitor, que lo hacen en la cola del semáforo `s`. Un proceso señalado ha de haber ejecutado `wait(s)` como consecuencia de ejecutar la operación de entrada en el monitor. Cuando el proceso señalado se reanude, lo hará heredando la exclusión mutua en el acceso al monitor que le cede el señalador, evitándose, de esta forma, los robos de señal.

```
entrada_al_monitor:   wait(s)  -- lo ejecutan todos los procesos para
                               -- asegurar la e.m. del monitor
urgente: contador de los procesos bloqueados
usem: semaforo para bloquear a los senialadores, inicialmente a 0
c.wait():
  cont_cond:= cont_cond + 1  -- numero de procesos bloqueados esperando
  if urgente > 0 -- hay procesos seniladores esperando
     then signal(usem)  -- libera a un senialador
     else signal(s)     -- levanta la exclusion mutua del monitor
  wait(semcond)         -- se bloquea esperando la senial cond
  cont_cond:= cont_cond - 1 -- un proceso bloqueado menos

c.signal():
  urgente:= urgente + 1 -- ahora hay 1 proceso mas que seniala
  if cont_cond > 0 then  -- hay procesos bloqueados esperando la senial cond
     begin
       signal(semcond); -- dejar entrar uno al monitor
       wait(usem);      -- se suspende como senialador
     end;
  urgente:= urgente -1  -- un proceso senialador bloqueado menos

salida_monitor:
  if urgente > 0        -- hay al menos un proceso senialador bloqueado
    then signal(usem)   -- despierta a un proceso senialador
    else signal(s)      -- levanta la e.m. del monitor
Inicialmente:
  inic(semcond,0); inic(usem, 0); urgente:= 0; cont_cond:= 0;
```

Figura 2.16: Implementación de las señales con semántica SU con semáforos

2.6.2 Implementación de las señales SU

Ahora los procesos que señalan no necesariamente han de terminar la ejecución del procedimiento del monitor, sino que se bloquean temporalmente en una cola de la máxima prioridad, entre las que se definen para la implementación de los monitores, ver figura 2.15. Cuando un proceso que ha sido desbloqueado de una cola de condición deja el monitor, porque salga o ejecute un nuevo `c.wait()`, ver figura 2.16, entonces ocurre que el proceso que lo desbloqueó tiene prioridad para entrar al monitor frente a otros nuevos procesos en la cola de entrada

del monitor. También podría ocurrir que el proceso desbloqueado señale a su vez y entonces podría entrar en la cola donde se encuentra el proceso que señaló anteriormente. La prioridad de los procesos bloqueados en la cola de señaladores, que viene representada en la siguiente implementación por el semáforo `usem`, se consigue haciendo que un proceso que ejecute la operación de salida o la operación `c.wait()` ceda el control del monitor al primero de los que esperan en la cola de `usem`, antes de dejar paso a los procesos que esperan en la cola de entrada, la cual está asociada al semáforo `s`.

2.7 Verificación de los monitores

En programas concurrentes que utilicen monitores se ha de realizar primero la demostracción de la corrección parcial de los procesos y de los monitores. Posteriormente, se aplicará la regla de la concurrencia para demostrar que los procesos secuenciales cooperan sin interferirse, así como que los invariantes de los monitores se mantienen durante toda la ejecución del programa.

La demostración de la corrección se realizará mediante la definición de invariantes globales de los datos que protege cada monitor. En lo que sigue nos vamos a centrar en realizar la demostración de la validez de los invariantes de los monitores, suponiendo que los procesos secuenciales del programa están bien programados individualmente y, por tanto, son correctos. También supondremos que no existe interferencia entre los procesos y los parámetros de los procedimientos de los monitores en la llamadas, ya que, a excepción de las variables permanentes de los monitores (no visibles a los procesos), sólo utilizaremos variables locales a los procesos de nuestro programa.

2.7.1 Axiomas y reglas de inferencia de los monitores

El programador de un monitor no puede conocer de antemano el orden en se llamarán sus procedimientos, por tanto, la demostración de la corrección parcial de estos ha de basarse en la verificación de una relación que se mantiene, denominada *invariante del monitor* (IM), y que establece una relación constante entre los valores permitidos de las variables permanentes del monitor. El IM ha satisfacerse antes de la ejecución de los procedimientos y cuando se deja libre de nuevo el acceso al monitor para otros procesos, esto es, al terminar la ejecución de un procedimiento o al ejecutar la operación de sincronización `c.wait()`.

Axioma de inicialización de las variables del monitor

El código de inicialización ha de ser programado como un procedimiento diferenciado, que se denomina *cuerpo* del monitor y ha de incluir la asignación de todas las variables permanentes del monitor, antes de que los procedimientos puedan ser llamados por primera vez por los procesos concurrentes.

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

Axioma de la operación `c.wait` para señales desplazantes

Los axiomas de la operación `c.wait()` y de la operación `c.signal()` sólo hacen referencia a los estados *visibles*[15] del programa, que vienen reflejados por los valores que toman las variables permanentes del monitor y los valores de las variables auxiliares que sea necesario definir en el texto de los procedimientos del monitor. Estos axiomas no tienen en cuenta el orden en el cual se desbloquean los procesos cuando se ejecuta la operación `c.signal()`. La ejecución de la operación `c.wait()` ocasiona, independientemente del tipo de señal con que se hayan programado los procedimientos, la cesión de la exclusión mutua del monitor[16], por lo tanto, se ha de asegurar que el IM es cierto antes de ejecutarla.

La operación `c.wait()` estará correctamente programada dentro de un procedimiento del monitor si se satisface el siguiente axioma:

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

Todas las señales con semática desplazante tienen definida la operación `c.signal()` que ocasiona la reanudación inmediata del proceso señalado que está esperando bloqueado en la cola de la variable condición `c`. Además, es una buena costumbre, cuando se programa con monitores, el asociar un predicado {C} con cada variable condición `c`, que indique la condición de *desbloqueo* de los procesos bloquedados en la cola de `c`; por ejemplo, en el caso del monitor *buffer*, la condición sería que el *buffer* no se encuentre lleno ($\{\text{C}: \text{n} \neq \text{N}\}$).

```
procedure retirar_mensaje (var x: tipo_mensaje)
 ...
 if nolleno.queue and (n <> N) then  -- la condicion de desbloqueo se satisface
    nolleno.signal;                  -- hay que se\~nalar a los procesos que esperan
 ...
```

El monitor mantiene el mismo estado desde que un proceso señala que la condición C se satisface hasta que el proceso señalado reanuda su ejecución. El predicado {L} indica un invariante definido sólo a partir de las variables locales del procedimiento.

Axioma de la operación `c.signal()` para señales desplazantes

La operación `c.signal()` ha de producir la reanudación inmediata de un proceso que está esperando bloqueado en la cola de la variable `c`. Debido a esto, toda condición definida a partir de las variables del monitor, que sea cierta antes de ejecutar `c.signal()`, mantendrá su valor lógico en el momento de reanudarse el proceso señalado. El predicado {C} de la siguiente regla de inferencia, que coincide con el predicado de la poscondición del axioma de `c.wait()`, representa a dicha condición:

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

[15] se excluyen los estados *no visibles* u ocultos del programa, tales como el orden de ejecución de los procesos que llaman a los procedimientos, el estado de las colas, etc.

[16] esto es, el permitir a otro proceso la entrada al monitor

Puesto que el proceso que reanuda su ejecución puede hacer falso posterioremente el predicado {C}, durante la ejecución del procedimiento del monitor, entonces sólo podemos suponer que se satisface el IM después de que la instrucción `c.signal()` vuelva. Obviamente, esto sólo podría ocurrir cuando el monitor se quede nuevamente libre. El que el proceso señalador sea elegido o no para continuar su ejecución cuando el monitor quede libre dependerá de la semántica de las señales que utilice el monitor y de los contenidos de las colas internas.

La operación `c.signal_all()` desbloquea a todos los procesos que se encuentren bloqueados en la cola de `c`. Sólo se reanudará uno de ellos, el resto ha de esperar a que el monitor quede nuevamente libre y no haya procesos esperando en una cola de mayor prioridad[17]. Esta operación tiene el mismo axioma que `c.signal()`.

La introducción de señales en la programación de monitores hace posible el escribir programas sujetos al riesgo de bloqueos, ya que nada nos asegura que el programador vaya a programar una operación `c.signal()` para reanudar a uno de los procesos bloqueados en colas de variables condición. Desde este punto de vista, si el mecanismo de envío de señales fuese realizado de manera automática[18] por el propio lenguaje de programación, cuando se detecta el estado adecuado del programa, se podrían evitar los despistes del programador que suelen dejar bloqueados incorrectamente a los procesos. Sin embargo, los lenguajes de programación con monitores no incluyen actualmente el mecanismo de *señalación automática de condiciones* porque su implementación suele ser muy ineficiente.

También podría ocurrir que la utilización indebida por el programador de una operación `c.wait()` en el texto de los procedimientos del monitor pudiera dejar bloqueado a un proceso del programa indefinidamente. Los métodos de demostración basados en asertos no pueden demostrar la ausencia de bloqueos basados en fallos de programación, tales como los anteriormente discutidos, ya que siempre se ha de suponer la terminación de la sentencia a la que se refieren los axiomas y las reglas de inferencia. Es responsabilidad del programador el evitar el riesgo de bloqueos en los programas debidos a una programación deficiente de la sincronización en los procedimientos de los monitores, así como el de otras deficiencias de diseño que pudieran afectar a la propiedad de vivacidad del programa durante la ejecución de los procesos. Se dice, por tanto, que los métodos de verificación basados en asertos sólo sirven para demostrar las propiedades de seguridad de los programas concurrentes.

2.7.2 Reglas de verificación de las señales SC

La ejecución de la operación `c.wait()` ocasiona que el proceso que se está ejecutando ceda el monitor y después se bloquee en la cola de la variable condición `c`. Puesto que se ha de dejar libre el monitor, para que otros procesos puedan entrar a ejecutar sus procedimientos, habría que asegurarse de la certeza del invariante del monitor, justo antes de programar esta operación en un procedimiento del monitor.

[17] como la cola de procesos que han señalado previamente

[18] por ejemplo, si el compilador utilizase semántica de señales automáticas

```
monitor semaforo;
 var s: integer; {IM : s ≥ 0}
     c: cond;
   procedure P;
     begin
      {IM}
      if s=0 then
        {s = 0 ∧ IM}
        c.wait;
      {s>0}
      else
      {s>0}
        null;
      {s>0}
      endif;
      {s > 0 }
      s:= s-1
      {s ≥ 0 → IM}
     end;

   procedure V;
     begin
      {IM}
       s:= s+1;
      {s>0}
       c.signal;
       {s≥ 0 → IM}
     end;
   begin
    {TRUE}
    s:= 0;
    {s ≥ 0} → {IM}
   end;
```

Figura 2.17: Verificación de un monitor suponiendo semántica desplazante de señales

Cuando el proceso que ejecutó `c.wait()` vuelva del bloqueo y antes de que entre nuevamente al monitor se ha de suponer que el invariante es cierto de nuevo, ya que, para que el monitor quede libre, el resto de los procesos deberían estar ejecutándose fuera del monitor, o esperando en la cola de entrada, o bloqueados en una cola de condición, en cualquiera de estas situaciones el IM se satisface. Los valores de algunas variables permanentes habrán cambiado, desde que el proceso se bloqueó, pero en el momento en que este reanude su ejecución, el conjunto de los valores de dichas variables satisfarán nuevamente el invariante del monitor. Este razonamiento nos lleva a proponer el siguiente axioma para la operación `c.wait`:

$$\{\text{IM} \wedge \text{L}\}\ \text{c.wait}\ \{\text{IM} \wedge \text{L}\}$$

La diferencia con el *wait* prioritario sólo consiste en el orden en el que los procesos se ordenan en la cola de la variable condición y, por tanto, en el orden en el que serán desbloqueados. Esto podría afectar a las propiedades de vivacidad, ya que podría inducir la inanición de algún proceso, pero nunca afectaría a la solidez[19] de la regla de verificación anterior en las demostraciones de propiedades de seguridad de los programas concurrentes.

Los procesos bloqueados en una cola de variable condición se desbloquean con la ejecución de las operaciones: `c.signal()`[20], `c.signal_all()`. En cualquiera de los dos casos, el proceso que señala continúa ejecutándose dentro del monitor, por tanto, ninguna variable permanente del monitor cambiará su valor como resultado de ejecutar una de estas operaciones,

[19] esto es, toda demostración realizada suponiendo señales con planificación FIFO es igualmente válida para señales prioritarias y a la inversa

[20] el lenguaje de programación Java, que define un mecanismo de señalación anónimo, llama a estas operaciones notify() y notifyAll()

luego tendrán el mismo axioma que el de la *sentencia* `null`: {P} `c.signal` {P}. La operación `c.signal_all()` difunde la señal a un grupo de procesos, desbloqueando a todos los procesos bloqueados en la cola de condición `c`. Tal como ocurre con la operación `c.signal`, no se modifica el valor de ninguna variable permamente del monitor y, por lo tanto, el axioma para ambas sentencias coincide.

2.7.3 Equivalencia entre los diferentes tipos de señales

Los diferentes tipos de señales pueden simularse unos a otros y poseen, por tanto, la misma capacidad para resolver los problemas de sincronización entre los procesos de un programa concurrente. La diferencia entre los diferentes mecanismos de señalación se basa en que unos permiten expresar una solución a un problema de sincronización de manera más sencilla que otros.

Si se cumplen determinadas situaciones, los diferentes tipos de señales se pueden intercambiar, sin que sea necesario la modificación del texto de los procedimientos del monitor. En estos casos, el monitor seguiría teniendo las mismas propiedades de seguridad, viéndose afectado sólo el orden de ejecución de los procesos y, quizás, la propiedad de *equidad* del programa. Naturalmente, para poder estar seguros de que ambos mecanismos se pueden intercambiar, hay que demostrar su equivalencia, es decir, que cualquier programa que utilice cualquiera de ellos mantiene las mismas propiedades concurrentes, lo cual se hace siguiendo el siguiente método constructivo:

Equivalencia entre señales SC y SA

Para probar que $SC \rightarrow^{simula} SA$ habría que considerar la posibilidad de definir una cola de condición c_B para cada condición `B` en la que tuvieran que esperar los procesos[21] que acceden al monitor. De acuerdo con esto, la operación `await(B)` de las señales automáticas podría ser simulada con un bucle que ejecutaría una operación de espera de las señales SC hasta que la condición `B` fuese cierta:

P_i	P_j
`...`	`...`
`await(B);`	`while NOT B do`
	`  c_B.wait;`
	`end do;`

Puesto que las señales SC son un mecanismo de señalación explícito, se tendría que incluir una llamada a la operación c_B`.signal` en aquellos puntos del texto de los procedimientos del monitor donde la condición `B` se evalúe como cierta. De esta forma se puede asegurar que los procesos en la cola de la condición c_B conseguirán alguna vez desbloquearse.

Las señales automáticas pueden simular a las señales SC ($SA \rightarrow^{simula} SC$). Para obtener una simulación de señales SC a partir de las señales SA, se puede declarar un array de booleanos,

[21] identificar las condiciones de bloqueo puede ser tedioso si B depende de parámetros y de variables definidas localmente en los procesos del programa, pero se puede hacer en cualquier caso

denominado `bloqueado[i]`, por cada variable condición del monitor con señales SC, donde `i : 1 ... n` es un índice que recorre los identificadores de los `n` procesos. De esta forma, sustituiríamos la declaración de una señal SC y las llamadas a las operaciones de sincronización en dicha señal:

```
                          Pi              Pj
        var             ---------       ---------
          c: cond;        ...             ...
                         c.wait;         c.signal
```

por el siguiente texto:

```
              Pi                                  Pj
 ------------------------------------        ----------
              ...                              ...
 insertar el identificador i del proceso       if not vacio(colac_c)
 al final de la colac_c                          k:= primero(colac_c)
 bloqueado[i]:= true;                          bloqueado[k]:= false;
 await(NOT bloqueado[i]);                      endif;
```

Equivalencia entre señales SW y SA

Las señales SW pueden simular a las señales automáticas ($SW \rightarrow^{simula} SA$). De una manera similar a la simulación realizada anteriormente con señales SC, la operación de espera `await(B)` de las señales automáticas se convierte en un bucle:

```
while NOT B do
  c_B.wait;
end do;
```

Ya que para salir del bucle anterior se ha de satisfacer la condición de desbloqueo `B`, sólo se necesita exigir el cumplimiento del IM después de la operación c_B.wait.

Se han de añadir operaciones c_B`.signal` en aquellos puntos del texto de los procedimientos del monitor donde la condición `B` se evalúe como cierta y el IM también lo sea, ya que las señales SW tienen una operación `c.signal` con semántica desplazante. Si se incluye un número suficiente de operaciones `c.signal` en los procedimientos adecuados del monitor, nunca se producirán bloqueos en el programa que sean debidos exclusivamente a la simulación.

Por otra parte, las señales SW se pueden simular con las señales automáticas ($SA \rightarrow^{simula} SW$), de una forma similar a la simulación realizada anteriormente para las señales SC. Aunque, en este caso, es necesario asegurarse de que el proceso desbloqueado por el envío de la señal se ejecuta inmediatamente, ya que la semántica de las señales SW es desplazante. Además, para garantizar la correcta implementación de la operación `c.signal`, los procesos que intentan entrar al monitor, antes de ejecutar el código del procedimiento que han llamado, han de ejecutar la operación `await(NOT señal_pendiente)` que simula la inserción de llamada en la cola de entrada[22] del monitor y sirve para evitar el *robo de señal*, que se daría si un proceso pudiera

[22]para las señales SW, en esta cola se han de bloquear también los procesos después de enviar una señal

entrar directamente al monitor adelantándose a otro proceso que acaba de ser señalado.

```
valores iniciales:
------------------
señal_pendiente: boolean:= false;
bloqueado: array[1..n] of boolean:= false;
```

```
     c.wait
   ----------
insertar el identificador i del
proceso al final de la colac_c
bloqueado[i]:= true;
await(NOT bloqueado[i]);
señal_pendiente:= false;
```

```
      c.signal
   ------------
if not vacio( colac_c)
   k:= primero(colac_c)
   señal_pendiente:= true;
   bloqueado[k]:= false;
   await(NOT señal_pendiente);
end if;
```

Equivalencia entre las señales SC y SW

Ambos tipos de señales son igual de potentes, en cuanto a su capacidad para resolver problemas de sincronización entre los procesos de un programa concurrente. De hecho, es fácil probar que se pueden simular entre sí, ya que, como hemos demostrado anteriormente, ambos tipos de señales son equivalentes a las señales automáticas (SA). Luego es fácil ver que se cumplen las siguientes relaciones de equivalencia:

$$SC \rightarrow^{simula} SA \rightarrow^{simula} SW$$

$$SW \rightarrow^{simula} SA \rightarrow^{simula} SC$$

Las señales SC y SW son en muchos casos directamente *intercambiables*, ya que un monitor *sintácticamente idéntico* puede utilizar un tipo u otro de señales sin que sus propiedades de seguridad se vean alteradas. Quizás la diferencia más significativa entre las señales SC y las señales SW consiste en que la operación de difusión de una señal a una cola de procesos no está definida para señales SW, ya que poseen una semántica desplazante, y sí lo está para las señales SC. Esto es debido a que con señales desplazantes no podemos estar seguros de que el siguiente bucle no degenere en un bucle de espera infinito, ya que los procesos tendrían que dejar libre el monitor cuando envían la señal, de acuerdo con la semántica desplazante, pero nada nos asegura que el proceso señalado no se pueda volver a bloquear. Si este escenario se repite indefinidamente, el bucle nunca terminaría, ya que siempre ocurriría que la cola asociada a la variable condición `c` nunca quedaría vacía.

```
while c.queue do
  c.signal;
end do;
```

Condiciones para intercambiar las señales SC y SW

El invariante del monitor ha de ser cierto antes de realizar cualquier operación `c.signal` y no es necesario que las variables permanentes satisfagan ninguna condición más fuerte, como postcondición de la operación `c.wait`, que el citado invariante. Esta condición asegura que se mantengan las mismas propiedades de seguridad del programa incluso si se intercambian ambos tipos de señales en el texto de un procedimiento del monitor, ya que las reglas de verificación de la operación `c.wait` para las señales SC y SW coinciden en ese caso.

Todas las llamadas a operaciones `c.signal` han de efectuarse antes de una operación `c.wait`, o bien como las últimas instrucciones del texto de un procedimiento del monitor. Además, las operaciones `c.signal` no pueden ser seguidas inmediatamente por sentencias de asignación que modifiquen el valor de las variables permanentes del monitor. El objetivo de esta regla es la de asegurar que la certeza de un predicado en el momento de ejecutar la operación `c.signal` se conserva hasta que el proceso señalado reanude su ejecución, tal como ocurriría si todas las señales tuviesen una semántica desplazante. Con señales SC no tendría por qué cumplirse esto, pero la regla establece que, después de ejecutar `c.signal`, no se puede modificar el valor de ninguna variable permanente del monitor, luego cualquier condición mantendrá su valor lógico por lo menos hasta que el proceso señalado se reanude.

Para mantener la equivalencia entre las señales, no se puede utilizar la operación de difusión de una señal `c.signal_all` a todos los procesos que se encuentran bloqueados en una cola de condición, ya que, como antes se dijo, esta operación no tiene una semántica bien definida para las señales desplazantes.

Como ejemplo de aplicación de las reglas anteriores, se puede ver en la figura 2.18 que el monitor del caso (a) funciona para señales SW, pero no funciona para señales SC, ya que `s` podría llegar a tomar valores negativos. También, se puede considerar el caso (b) que funciona para señales SC, pero produce interbloqueo para señales SW.

2.8 El problema de la anidación de llamadas en monitores

La adquisición y liberación de la exclusión mutua es un problema no-trivial si se permite que las llamadas a los monitores se puedan anidar. Por ejemplo, suponer que el *proc1* del monitor *mon1* llama al procedimiento *proc2* del monitor *mon2*. Si *proc2* contiene una operación *wait* se presenta la disyuntiva entre 2 opciones: (a) levantar la exclusión en los dos monitores, (b) levantar la exclusión sólo en *mon2*.

Si primero consideramos la opción (b), tendremos como consecuencia que todos los monitores que han sido llamados antes de *mon2* resultan inaccesibles para el resto de los procesos. El efecto de esta situación es una pérdida de eficiencia de la aplicación; incluso, podría llevar a una situación de interbloqueo si la reanudación del proceso que llamó a `wait` depende de otro proceso que deba realizar la misma secuencia de llamadas a monitores antes de efectuar la operación `signal` que lo desbloquearía.

```
(a) Monitor semaforo_FIFO;
     {IM: s>= 0 }
     var c: cond;
     s: integer;
     procedure P;
       begin
        if (s= 0) then c.wait;
        {s > 0}
        s:= s-1;
       end do;
     procedure V;
       begin
        s:= s+1;
        c.signal;
       end;
     begin
       s:=0;
     end;
```

```
(b) Monitor semaforo_FIFO2;
      {IM: s>=0}
      ...

    procedure P;
      begin
       while s= 0 do
         c.wait;
         {s>= 0}
       end do;
       {s>0}
       s:= s-1;
      end;
    procedure V;
       begin
         c.signal;
         s:= s+1;
       end;...
```

Figura 2.18: Implementaciones alternativas de semáforo FIFO con monitores

La opción (a), levantar la exclusión mutua de todos los monitores en la secuencia de llamadas, necesita implementar una manera de *recordar* en qué monitores había entrado ya el proceso cuando hizo la llamada a `wait`. La implementación mediante una pila que almacene las llamadas parece ser lo más obvio. Sin embargo el conseguir que el proceso bloqueado en el último monitor recupere la exclusión mutua de todos los monitores anteriores puede resultar difícil de implementar, ya que en dichos monitores pueden existir ahora otros procesos. Una implicación adicional de esta opción es que el invariante del *mon1* debe de restablecerse antes de que se produzca la llamada del *mon2*, lo cual puede no ser siempre fácil de conseguir.

El análisis anterior muestra que ambas opciones tienen serias desventajas. Una forma expeditiva de evitarlas es implementar un mecanismo único de exclusión *global* para todos los monitores, más que el implementar un mecanismo de exclusión *local* para cada monitor separadamente. Pero si se utiliza un mecanismo global de exclusión, entonces desaparece el problema de las llamadas anidadas. Por supuesto, la ganancia en simplicidad de esta solución no es sin coste, ya que el grado de paralelismo potencial del sistema es reducido artificialmente.

Otra forma de *"resolver"* el problema sería el prohibir las llamadas anidadas a los monitores. Pero esto constituye una restricción demasiado severa, que sería inaceptable para cualquier sistema construido jerárquicamente.

Una solución al problema, que ha sido implementada en el lenguaje de programación *Concurrent Pascal* de P. Brinch-Hansen [Hansen, 1977], define un mecanismo de *exclusión local* para cada monitor, siguiendo la opción (b) anterior.

Por último, debe ser reseñada la propuesta de Parnas que considera a los monitores "sólo como una herramienta de estructuración de recursos compartidos que están sujetos a acceso concurrente" y, por tanto, no cree que la anidación de llamadas a los monitores tenga por qué

llevar asociada una única regla, ya que hay casos en los que los procedimientos de un monitor pueden ejecutarse concurrentemente sin efectos adversos, o en los que el invariante del monitor puede establecerse fácilmente antes de que se realice una llamada anidada. Esta propuesta, define los monitores dando libertad para especificar el que ciertos procedimientos se ejecuten concurrentemente y que la exclusión mutua se libere para ciertas llamadas, pero no para otras. Todos estos conceptos han sido implementados en el lenguaje de programación Mesa.

2.9 Problemas resueltos

Ejercicio 1

Un algoritmo para el cual sólo pudiésemos demostrar que cumple las 4 condiciones de Dijkstra, qué tipo de propiedades concurrentes satisfaría: a) seguridad, b) vivacidad, c) equidad?

Solución:

- *Seguridad*– Para que un algoritmo cumpla la propiedad de seguridad no puede existir interbloqueo ni más de un proceso en sección crítica al mismo tiempo. Sabiendo que el algoritmo que nos ocupa cumple las cuatro condiciones de Dijkstra, podemos asegurar que no habrá interbloqueo (4 condición) y que no habrá más de un proceso en sección crítica a la vez. Podemos concluir por tanto en que el algoritmo es seguro.
- *Vivacidad*– Con las cuatro condiciones de Dijkstra se garantiza la exclusión mutua en el acceso a la sección crítica por parte de los procesos y también la alcanzabilidad de la sección crítica (no interbloqueo), pero no evita el riesgo de inanición de los procesos. Por lo tanto, puesto que no se puede asegurar que nunca se produzca inanición de los procesos, no se cumple la propiedad de vivacidad.
- *Equidad*– Para que se cumpla la propiedad de equidad en un algoritmo, todos los procesos que lo ejecutan concurrentemente han de hacerlo consiguiendo avanzar en la ejecución de sus instrucciones de una forma justa. En caso de inanición esta propiedad no se cumple, pues un proceso puede no alcanzar nunca la sección crítica si los otros procesos siempre lo adelantan.

Ejercicio 2

¿Qué ocurriría si el algoritmo de Dekker se hubiera programado como se muestra?

```
Proceso Pi;
BEGIN
REPEAT
    Turno:=i;
    WHILE Turno<>i DO ;
    (*Seccion Critica*)
    Turno:=j;
    FOREVER
END;
```

Figura 2.19: Implementación alternativa del algorítmo de Dekker

Solución:

No cumpliría la propiedad de seguridad, pues puede acceder más de un proceso a SC a la vez. Un ejemplo de escenario es el siguiente: "Un proceso P_i pone $Turno := i$, entrando en SC directamente. Mientras este está en SC, otro proceso P_j pide acceder también a ella, poniendo $Turno := j$. Al igual que P_i accedería, estando pues dos procesos en SC a la vez".

Tampoco cumpliría la propiedad de vivacidad, ya que los procesos podrían sufrir inanición. El escenario sería el siguiente: "Un proceso P_i, quiere acceder a SC asignando $Turno := i$. Si hay otro proceso P_j muy rápido y repetitivo que ponga $Turno := j$ antes de que P_i pueda comprobar la condición del while, siempre entrará P_j en SC, esperando P_i en el bucle interno".

Ejercicio 3

Al siguiente algoritmo se le conoce como solución de Hyman al problema de la exclusión mutua para dos procesos. ¿Es correcta dicha solución?

```
        Proceso Pi
        ci:=0;
        while Turno <> i do
        begin
            while cj=0 do nothing;
            Turno:=i;
        end;
        (*Seccion Critica*)
        ci:= 1;
(Inicialmente: c1,c2:=1,turno:=1)
```

Figura 2.20: Solución de Hyman al problema de la exclusión mutua

Solución:

Esta solución no es segura, pues puede ocurrir lo siguente: "Un proceso P_i está accediendo a SC (suponemos que $Turno = i$). En este instante otro proceso P_j pide entrar en SC quedándose en el bucle interior a la espera de que P_i finalice su ejecución. Una vez finalizada ésta, P_i pone $c1 := 1$, con lo que P_j sale de este bucle interno, pero antes de que P_j haga $Turno := j$, un proceso P_i, que acaba de llegar, comprueba la condición del while externo pasando a ejecutar la SC (pues aún $Turno = i$). En este momento P_j cambia $Turno := j$ saliendo pues del bucle externo y accediendo también a la SC (ya está en SC P_i)".

Ejercicio 4

Se tienen 2 procesos concurrentes que representan 2 máquinas expendedoras de tickets (señalan el turno en que ha de ser atendido el cliente). Los números de tickets se representan mediante los valores que toman 2 variables, asignadas inicialmente a 0. El proceso con el número de ticket más bajo entra en su sección crítica. En caso de tener dos números iguales se procesa primero al proceso número 1. Demostrar a) que los valores de las variables $ni = 1, 1 <= i <= 2$, son necesarios, y b) que se verifican las propiedades de vivacidad de la solución.

```
PROGRAM PanaderiaDeLamport;

(1)VAR n1,n2:INTEGER;
(2)PROCEDURE P1;       PROCEDURE P2;
(3)BEGIN               BEGIN
(4) REPEAT              REPEAT
(5)     n1:=1;              n2:=1;
(6)     n1:=n2+1;           n2:=n1+1;
(7)     WHILE (n2<>0)AND    WHILE (n1<>0)AND
(8)     (n2<n1) DO nothing; (n1<=n2) DO nothing;
(9)      (*Region Critica*)  (*Seccion Critica*)
(10)      n1:=0;              n2:=0;
          ...                 ...
          FOREVER             FOREVER
   END;                END;
BEGIN n1:=0;n2:=0;
COBEGIN P1;P2 COEND;
END.
```

Figura 2.21: Programa de la Panadería de Lamport

Solución:

a) Este algoritmo presenta 2 escenarios en los cuales no se cumple la propiedad de exclusión mutua: $n1 = 0, n2 \geq n1$ y $n2 = 0, n1 > n2$. Si los valores de $n1 = n2 = 0$ antes de ejecutar las instrucciones número 5, entonces se podría dar alguno de los escenarios indicados anteriormente por entrelazamiento de las instrucciones $n1 = n2 + 1$ y $n2 = n1 + 1$. Sin embargo, si los valores iniciales de $n1 = n2 = 1$ antes de ejecutar las instrucciones número 5, entonces nunca se puede dar, en ningún entrelazamiento, los valores anteriormente mencionados.

b) Si $P1$ está en SC, entonces $n1 \leq n2$. Cuando $P1$ salga de SC, asignará su clave a cero ($n1 = 0$) y suponiendo que es un proceso repetitivo y muy rápido, entonces ejecutará $n1 = n2 + 1$ que hará que $n1 > n2$ y por tanto que $P1$ se detenga en el bucle while (instrucción número 7) avanzando así $P2$.

Ejercicio 5

El siguiente programa es una solución al problema de la exclusión mutua para 2 procesos. Discutir la corrección de esta solución. Si fuera correcta, entonces demostrarlo. Si no lo fuese, escribir escenarios (tablas con los valores de las variables relevantes) que demuestren la incorrección.

```
PROGRAM intento; (Inic.c1,c2:=1)

VAR c1,c2:INTEGER:

PROCEDURE P1;          PROCEDURE P2;
BEGIN                  BEGIN
    REPEAT                 REPEAT
        REM1;                  REM2;
    REPEAT                 REPEAT
        c1:=1 c2;              c2:=1 c1;
    UNTIL c2<>0;           UNTIL c1<>0;
    CRIT1;                 CRIT2;
    C1:=1;                 C2:=1;
    FOREVER;               FOREVER;
END;                   END;
BEGIN
    c1:=1;c2:=1;
    COBEGIN P1;P2 COEND
END.
```

Figura 2.22: Solución al problema de la exclusión mutua para 2 procesos

Solución:

Este algoritmo incumple las propiedades de ausencia de interbloqueo $c1 = c2 = 0$ y de exclusión mutua $c1 = c2 = 1$, ambas situaciones se darían con ciertos entrelazamientos de las siguientes instrucciones:

La secuencia de instrucciones que ejecuta cada uno de los procesos es:

En el caso de P1:

```
c1:= 1 - c2     // en ensamblador //          Ac = -c2
c1:= 1          //      -->       //          Ac = Ac + 1
                                              c1 = Ac
```

En el caso de P2:

```
c2:= 1 - c1     // en ensamblador //          Ac = -c1
c2:= 1          //      -->       //          Ac = Ac + 1
                                              c2 = Ac
```

A continuación se muestran con más detalle los entrelazamientos de instrucciones que producen interbloqueo y violación de la exclusión mutua:

Proceso	C1	C2	Acumulador
Inicialmente	1	1	
P1	1	1	-1
P2	1	1	-1
P1	1	1	0
P2	1	1	1
P1	1	1	1
P2	1	1	1

Figura 2.23: Violación de la exclusión mutua

Proceso	C1	C2	Acumulador
Inicialmente	1	1	
P1	1	1	-1
P1	1	1	0
P2	1	1	-1
P2	1	1	0
P1	0	1	0
P2	0	0	0

Figura 2.24: Interbloqueo

Ejercicio 6

Con respecto al algoritmo de Peterson para n procesos: ¿Sería posible que llegaran 2 procesos a la etapa $n-2$, 0 procesos en la etapa $n-3$ y en todas las etapas anteriores existiera al menos 1 proceso?

						SC
Etapas	**0**	**1**	**.....**	**n-3**	**n-2**	**n-1**
Nº Procesos	1	1	1.....	0	2	0

Figura 2.25: Supuesto de ejecución sobre el algorítmo de Peterson para n procesos.

Solución:

Suponemos que el primer proceso en llegar a la etapa $n-2$ ha llegado porque precedía a todos los demás (lema 1). Para que llegue un segundo proceso a esta etapa, ha de cumplir alguna de las condiciones siguientes (lema 2):

- Precede a todos los demás: este no es el caso, pues hay ya un proceso en la etapa $n-2$.
- No está sólo en la etapa $n-3$: tampoco es esto correcto, pues el escenario que se nos presenta tiene 0 procesos en la etapa $n-3$.

Por tanto podemos decir que este escenario no es posible.

Ejercicio 7

Demostrar que incluso si suponemos una implementación de los monitores con semáforos FIFO, las colas condición del monitor que resultan de dicha simulación con semáforos no cumplen con la propiedad FIFO cuando un proceso es desbloqueado mediante la ejecución de la operación *c.signal* por parte de otro proceso.

```
    c.wait()
--------
(1) cont_cond++;
(2) signal(s);
(3) wait(sem_cond);
(4) cont_cond--;
```

Solución:

La ejecución de las operaciones (2) y (3) no se realiza atómicamente; luego un proceso después de ejecutar (2)$signal(s)$ podría no bloquearse en (3)*wait* y adelantarse a procesos ya bloqueados en *sem_cond* cuando el monitor se queda libre de nuevo después de que otro proceso señale la condición y salga.

Ejercicio 8

Se consideran dos recursos denominados r1 y r2. Del recurso r1 existen N1 copias y del recurso r2 existen N2. Escribir un monitor que gestione la asignación de los recursos de los procesos, suponiendo que cada uno de estos puede pedir:

- **Un ejemplar del recurso r1**
- **Un ejemplar del recurso r2**
- **Un ejemplar del recurso r1 y otro del recurso r**

La solución deberá satisfacer estas dos condiciones:

1. **Un recurso no será asignado a un proceso que demande un ejemplar de r1 o un ejemplar de r2 hasta que al menos un ejemplar de dicho recurso quede libre.**
2. **Se dará prioridad a los procesos que demanden un ejemplar de ambos recursos.**

Solución:

La solución sería:

```
Monitor Pedir_Recurso(){

    procedure solicitar_recurso1(){
        if (n1 = 0) c1.wait();
        n1--; //Asignado r1
    }

    procedure solicitar_recurso2(){
     if (n2 = 0) c2.wait();
     n2--;//Asignado r2
    }

    procedure solicitar_r1_r2 () {
        if ((n1 = 0) || (n2 = 0)) c3.wait();
        n1--;
        n2--;
    }

    procedure ceder_r1(){
        n1++;
        if ((c3.queue()) && (n2 >0)) c3.signal();
        else c1.signal();
    }

    procedure ceder_r2(){
        n2++;
        if((c3.queue()) && (n1>0)) c3.signal();
        else c2.signal();
    }

    Begin
     n1,n2: integer;
     c1,c2,c3: cond;
    End
}
```

Ejercicio 9

Programar los procedimientos de un monitor necesarios para simular la operación abstracta de sincronización denominada cita entre 2 procesos: el primer proceso preparado para sincronizarse ha de esperar hasta que el otro esté también preparado para establecer dicha sincronización.

Solución:

```
Monitor Cita(){
    procedure llama_cita(){
     c2.signal ();
     if ! senialado
     c1.wait();
        senialado=FALSE;
    }

    procedure cumple_cita(){
     if ! c1.queue
     c2.wait();
        senialado=TRUE;
     c1.signal();
    }

    Begin
        bool senialado = FALSE;
    c1,c2 : cond;
    End
}
```

En el procedimiento *llama_cita*() el proceso A llama para ver si está ya B y hacer la cita. Si B no levanta la bandera (es decir, aún no está) se espera.

En el procedimiento *cumple_cita*() el proceso B se "asoma" para ver si ya ha llegado A, si no ha llegado se espera. Levanta la bandera para indicar al otro que hay cita.

Ejercicio 10

Suponer un garaje de lavado de coches con dos zonas: una de espera y otra de lavado de coches con 100 plazas de lavado. Un coche entra en la zona de lavado de garaje sólo si hay (al menos) 1 plaza libre; si no, se queda en la cola de espera. Si un coche entra en la zona de lavado, busca una plaza libre y espera hasta que es atendido por un empleado del garaje. Los coches no pueden volver a entrar al garaje hasta que el empleado que les atendió les cobre el servicio. Suponemos que hay más de 100 empleados que lavan coches, cobran, salen y vuelven a entrar al garaje. Cuando un

empleado entra en el garaje comprueba si hay coches esperando ser atendidos (ya situados en su plaza de lavado; si no, aguarda a que haya alguno en esa situación. Si hay al menos un coche esperando, recorre las plazas de la zona de lavado hasta que lo encuentra, entonces lo lava, le avisa de que puede salir y, por último, espera a que le pague. Puede suceder que varios empleados hayan terminado de lavar sus coches y estén todos esperando el pago de sus servicios, pero no se admite que un empleado cobre a un coche distinto del que ha lavado. También hay que evitar que al entrar 2 ó más empleados a la zona de lavado, habiendo comprobado que hay coches esperando, seleccionen a un mismo coche para lavarlo. Se pide: programar una simulación de la actuación de los coches y de los empleados del garaje, utilizando un monitor con señales urgentes y los procedimientos que se dan a continuación. Se valorará más una solución al problema anterior que utilice un número menor de variables *signal*. Los procedimientos a implementar en el monitor son:

- *procedure lavado_de_coche*; – **lo ejecutan los coches, incluye la actuación completa del coche desde que entra al garaje hasta que sale; se supone que cuando un coche espera deja a los otros coches que se puedan mover dentro del garaje.**
- *procedure siguiente_coche*() : *plazas_libres*; – **es ejecutado por los empleados. Devuelve el número de plaza donde hay un coche esperando ser lavado.**
- *procedure terminar_y_cobrar*(*i* : *plazas_libres*); – **es ejecutado por los empleados, para avisar a un coche que ha terminado su lavado; termina cuando se recibe el pago del coche que ocupa la plaza "i".**

Solución:

La solución al problema planteado sería la siguiente:

```
Monitor Garaje(){
    procedure lavado_de_coche(i:1  100){
        if ( pl_libre = 0 ) entrada.wait();
        pl_libres--;
        esperando++;
        terminado[i] = false;
        comenzar.signal();
        while !(terminado[i]){
            lavado.signal();
            lavado.wait();
        }
        pl_libres++;
        pagado[i] = TRUE;
        puerta.signal();
        entrada.signal();
    }
```

```
    procedure terminar_y_cobrar(i:1  100){
     terminado[i] = TRUE;
     lavado.signal();
     while !(pagado[i]){
         puerta.signal();
         puerta.wait();
        }
        pagado[i]=TRUE;
    }

    int siguiente_coche(i:1  100) {
     int k =0;
     if(esperando=0) comenzar.wait();
     esperando--;
     do{
         k++;
        }while( !terminado[k]);
        return k;
    }

    Begin
        pl_libres,esperando: int = 0;
        bool terminando[100]: bool = TRUE;
        bool pagado[100]: bool = FALSE;
        entrada, lavado, puerta, comenzar: cond;
    End
}
```

2.10 Problemas propuestos

1. Supongamos el algoritmo de exclusión mutua que expresamos a continuación. Tenemos los procesos: 0, , ... n − 1. Cada proceso i tiene una variable $s[i]$, inicializada a 0, que puede tomar los valores 0/1. El proceso i puede entrar en la sección crítica si:

$$s[i] \; != \; s[i-1] \;\; \textit{para } i > 0;$$
$$s[0] \;\; = \;\; s[n-1] \;\; \textit{para } i = 0;$$

   Tras ejecutar su sección crítica, el proceso i deberá hacer:

$$s[i] \;\; = \;\; s[i-1] \;\; \textit{para } i > 0;$$
$$s[0] \;\; = \;\; (s[0]+1) \; mod \; 2 \;\; \textit{para } i == 0;$$

2. Algunos ordenadores antiguos tenían una instrucción llamada TST (TestAndSet). Existe una variable global `c` en el sistema, en memoria común a los procesadores, llamada código de condición. Ejecutando la instrucción `TST(l)` para la variable local `l` se obtiene lo mismo que con las 2 instrucciones siguientes:
   - `l=c;`
   - `c=1;`

   (a) Discutir la corrección de la solución al problema de la exclusión mutua del algoritmo de la figura 2.26.

   (b) ¿Qué ocurriría si la instrucción TST fuese reemplazada por las 2 instrucciones anteriores?

3. La instrucción EX intercambia los contenidos de 2 posiciones de memoria. `EX(a,b)` es equivalente a una ejecución indivisible de las 3 operaciones siguientes:

```
temp=a;
a=b;
b=temp;
```

   (a) Discutir la corrección de la solución para el algoritmo de exclusión mutua de la figura 2.27.

   (b) ¿Qué ocurriría si la instrucción primitiva EX fuese reemplazada por las 3 instrucciones anteriores?

4. Con respecto al algoritmo de la figura 2.28 (algoritmo de Dijkstra para N procesos), demostrar la falsedad de la siguiente proposición: *si un conjunto de procesos está intentando pasar simultáneamente el primer bucle(5), y el proceso que tiene el turno está* pasivo*, entonces siempre conseguirá entrar primero en sección crítica el proceso de dicho grupo que consiga asignar la variable* turno *en último lugar.*

5. El algoritmo de la figura 2.29 (algoritmo de Knuth para N-procesos) resuelve el problema de la exclusión mutua para N-procesos, para lo cual utiliza N variables booleanas,
   `flag: array[ 0..N - 1 ] of ( solicitando, enSC, pasivo );`
   una variable `turn: 0..n - 1` y la variable local `j`.
   (a) Demostrar que el algoritmo de Knuth verifica todas las propiedades exigibles a un programa concurrente, incluyendo la de equidad.

```
int main (int argc, char *argv[])
{
int c=0;
void *p1(void *arg){  void *p2 (void *arg){
  int l;                int l;
  do{                   do{
  //resto inst.         //resto inst.
    do{                   do{
       TST(l);               TST(l);
    }while(l==0);         }while(l==0);
    //Seccion critica     //Seccion critica
    c=0;                  c=0;
    }while(true);       }while(true);
 }                     }
cobegin p1(); || p2(); coend,
}
```

Figura 2.26: Problema 2.

```
int main (int argc, char *argv[])
{
int c=1;
void *p1(void *arg){  void *p2 (void *arg){
  int l=0;              int l=0;
do{                   do{
  //resto inst.         //resto inst.
    do{                   do{
       EX(c,l);              EX(c,l);
    }while(l!=1);         }while(l!=1);
    //Seccion critica     //Seccion critica
    EX(c,l);              EX(c,l);
    }while(true);       }while(true);
 }                     }
cobegin p1(); || p2(); coend,
}
```

Figura 2.27: Problema 3.

(b) Escribir un escenario en el que 2 procesos consiguen pasar el bucle de la instrucción (5), suponiendo que el turno lo tiene inicialmente el proceso `p(0)`.

6. Si en el algoritmo de la figura 2.28 se cambia la instrucción (6) por esta otra: `if flag[turno] !=SC`, entonces el algoritmo dejaría de ser correcto. Indicar qué propiedad(es) de corrección fallaría(n) y justificar por qué.

7. Si en el algoritmo de la figura 2.29 se hacen las siguientes sustituciones:
   - La condición de la instrucción `until` de (15) por la condición: `(j >= N)and( turno=i or flag[turno]= pasivo)`
   - Se inserta el siguiente bucle después de la instrucción (17):

```
while (j !=turn )and( flag[j]=pasivo) do
(19)    j := j + 1;
(20) enddo;
```

(a) Verificar las propiedades de exclusión mutua, alcanzabilidad de la sección crítica, vivacidad y equidad del algoritmo.

(b) Calcular el número de turnos máximo que puede llegar a tener que esperar un proceso que quiera entrar en su sección crítica con el algoritmo anterior.

```
var turn :0..N-1;
flag: array [0 .. N-1] of (pasivo,
solicitando, enSC);
flag:= pasivo;
P(i)::
(1)   <resto instrucciones>
(2)   repeat
(3)      flag[i] := solicitando;
(4)      j := turn;
(5)      while (turno !=i) do
(6)         if (flag[turno] =  pasivo) then
(7)            turn:=i;
(8)         endif;
(9)    enddo;
(10)    flag[i] := enSC;
(11)    j := 0;
(12)    while ( j < N )and
        ( (j = i)or flag[j]!= enSC )do
              j := j + 1;
(14)    enddo;
(15) until (j >= N);
<<seccion critica>>
(16) flag[i] := pasivo;
```

Figura 2.28: Problema 4.

```
var flag: array [0 .. N-1] of (pasivo,
solicitando, enSC);
flag:= pasivo;
turn := 0;
P(i)::
(1)   <resto instrucciones>
(2)   repeat
(3)      flag[i] := solicitando;
(4)      j := turn;
(5)      while (j !=i) do
(6)         if flag[j] !=  pasivo then
(7)            j := turn
(8)         else j := ( j - 1 ) mod N;
(9)         endif;
(10)    enddo;
(11)    flag[i] := enSC;
(12)    j := 0;
(13)    while ( j < N )and
        ( (j = i)or flag[j]!= enSC )do
              j := j + 1;
(14)    enddo;
(15) until (j >= N);
(16) turn := i;
<<seccion critica>>
(17) j := ( turn + 1 ) mod N;
(18) turn := j;
(19) flag[i] := pasivo;
```

Figura 2.29: Problema 5.

8. Aunque un monitor garantiza la exclusión mutua, los procedimientos de los módulos *monitor* tienen que ser re-entrantes. Explicar por qué.

9. Demostrar que incluso si suponemos una implementación de los monitores con semáforos FIFO, las colas de las *variables condición* del monitor que resultan de dicha simulación con semáforos no cumplen con la propiedad FIFO cuando un proceso es desbloqueado mediante la ejecución de la operación `c.signal()` por parte de otro proceso.

10. Dos tipos de personas, representados por los tipos de procesos A y B, entran en una habitación. La habitación tiene una puerta muy estrecha por la que cabe sólo 1 proceso. La actuación de los procesos es la siguiente:

(a) Un proceso de tipo A no puede abandonar la habitación hasta que encuentre a 10 procesos de tipo B.

(b) Un proceso de tipo B no puede abandonar la habitación hasta que no encuentre a un proceso de tipo A y otros 9 procesos de tipo B.

Siempre se ha de cumplir la condición: en la habitación no hay procesos de tipo A o hay menos de 10 procesos de tipo B. Si en algún momento no se cumple dicha condición, tendrían que salir 1 proceso de tipo A y 10 procesos de tipo B para que se volviera a cumplir la condición. Cuando los procesos salen de la habitación, no pueden entrar nuevos procesos a la habitación (de ningún tipo) hasta que salgan todos. Implementar un monitor para solucionar el problema.

Figura 2.30: Representación gráfica de la barbería

11. En un pueblo hay una pequeña barbería con una puerta de entrada y otra de salida, dichas puertas son tan estrechas que sólo permiten el paso de una persona cada vez que son utilizadas. Como se puede ver en la figura 2.30, en la barbería hay un número indeterminado (nunca entran los suficientes clientes para que se agoten los asientos libres) de sillas de 2 tipos:
    (a) sillas donde los clientes esperan a que entren los barberos,
    (b) sillas de barbero donde los clientes esperan hasta que termina su corte de pelo.

    No hay espacio suficiente para que más de una persona (barbero o cliente) pueda moverse dentro de la barbería en cada momento, p.e., si los clientes se dan cuenta que ha entrado un barbero, entonces sólo 1 cliente puede levantarse y dirigirse a una silla de tipo (b); un barbero, por su parte, no podría moverse para ir a cortarle el pelo hasta que el cliente se hubiera sentado. Cuando entra un cliente en la barbería, puede hacer una de estas 2 cosas:
    - Aguarda en una silla de tipo (a) y espera a que existan barberos disponibles para ser atendido, cuando sucede esto último, el cliente se levanta y vuelve a sentarse, esta vez en una silla de tipo (b), para esperar hasta que termine su corte de pelo.
    - Se sienta directamente en una silla de tipo (b), si hay barberos disponibles.

    Un cliente no se levanta de la silla del barbero hasta que éste le avisa abriéndole la puerta de salida. Un barbero, cuando entra en la barbería, aguarda a que haya clientes sentados en una silla de tipo (b) esperando su corte de pelo. Después revisa el estado de los clientes, siguiendo un orden numérico establecido, hasta que encuentra un cliente que

espera ser atendido, cuando lo encuentra comienza a cortarle el pelo y él mismo pasa a estar ocupado. Cuando termina con un cliente, le abre la puerta de salida y espera a que el cliente le pague, después sale a la calle para refrescarse. El barbero no podrá entrar de nuevo en la barbería hasta que haya cobrado al cliente que justamente acaba de atender. No se admite que un cliente pague a un barbero distinto del que cortó el pelo.

Resolver el problema anterior utilizando un solo monitor que defina los siguientes procedimientos:

```
procedure corte_de_pelo ( i: numero_de_cliente );
(* lo ejecutaran los clientes, incluye la actuacion completa del cliente,
desde que entra en la barberia hasta que sale; se supone que cuando
un cliente espera en una silla deja la barberia libre y otra persona
puede moverse dentro de ella*);

function siguiente_cliente ( ): numero_de_cliente;
(* es ejecutado por los barberos. Devuelve el numero de cliente
seleccionado,el criterio de seleccion empleado consiste en revisar el
estado de los clientes hasta encontrar un cliente que espera ser atendido *)

procedure termina_corte_de_pelo ( i: numero_de_cliente ) ;
(* es utilizado por los barberos. El parametro sirve para que el barbero
sepa a que cliente tiene que cobrar  *)
```

12. Demostrar que el monitor "productor-consumidor" es correcto utilizando las reglas de corrección de la operación c.wait() y c.signal() para señales desplazantes. Considerar el siguiente invariante global: I == $\{ 0 \leq n \leq \text{MAX} \}$.

13. Escribir una solución al problema de lectores-escritores con monitores otorgando la prioridad a los procesos escritores.

14. Para cada uno de los esquemas de prioridad en el problema de lectores-escritores, intentar pensar una aplicación en la que es razonable el esquema de prioridad correspondiente.

15. Demostrar la corrección de un monitor que implemente las operaciones de acceso al buffer circular para el problema del productor-consumidor utilizando el siguiente invariante:

$$\begin{aligned} 0 &\leq n \leq N; \quad \textit{No hay procesos bloqueados} \\ 0 &> n; \quad \textit{Hay} \mid n \mid \textit{consumidores bloqueados} \\ n &> N; \quad \textit{Hay}\,(n-N)\,\textit{productores bloqueados} \end{aligned}$$

Escribir un monitor que cumpla el invariante anterior, es decir:

- Se haga `novacio.signal()` solamente cuando haya consumidores bloqueados.
- Se haga `nolleno.signal()` solamente cuando haya productores bloqueados.

Se supone que el buffer tiene N posiciones y se utilizan las dos señales mencionadas anteriormente. No está permitido utilizar la operación `c.queue()` para saber si hay procesos bloqueados en alguna cola de variable condición.

16. Coches que vienen del norte y del sur pretenden cruzar un puente sobre un río. Sólo existe un carril sobre dicho puente. Por tanto, en un momento dado, sólo puede ser cruzado simultáneamente por uno o más coches en la misma dirección (pero no en direcciones opuestas).

(a) Completar el código del siguiente monitor que resuelve el problema del acceso al puente suponiendo que llega un coche del norte (sur) y cruza el puente si no hay otro coche del sur (norte) cruzando en ese momento.

(b) Mejorar el monitor anterior, de forma que la dirección del tráfico a través del puente cambie cada vez que lo hayan cruzado 10 coches en una dirección, mientras 1 ó más coches estuviesen esperando cruzar el puente en dirección opuesta.

17. Una tribu de antropófagos comparte una olla en la que caben M misioneros. Cuando algún salvaje quiere comer, se sirve directamente de la olla, a no ser que ésta esté vacía. En este último caso, el salvaje despertará al cocinero y esperará a que haya rellenado la olla con otros M misioneros.

```
proceso Salvaje          proceso Cocinero;
repetir                  repetir
  servirse_1_misionero;    dormir;
  comer;                   rellenar_olla;
siempre;                 siempre;
```

Implementar un monitor para la sincronización requerida, teniendo en cuenta que:

(a) La solución no debe producir interbloqueo.

(b) Los salvajes podrán comer siempre que haya comida en la olla.

(c) Solamente se despertará al cocinero cuando la olla esté vacía.

18. Una cuenta de ahorros es compartida por varias personas/procesos. Cada persona puede depositar o retirar fondos de la cuenta. El saldo actual de la cuenta es la suma de todos los depósitos menos la suma de todos los reintegros. El saldo nunca puede ser negativo.

(a) Programar un monitor para resolver el problema, todo proceso puede retirar fondos mientras la cantidad solicitada c sea menor o igual que el saldo disponible en la cuenta en ese momento. Si un proceso intenta retirar una cantidad `c` mayor que el saldo, debe quedar bloqueado hasta que el saldo se incremente lo suficiente para que se pueda atender la petición, como consecuencia de que otros procesos depositen fondos en esa cuenta. El monitor debe tener 2 métodos: `depositar(c)` y `retirar(c)`. Suponer que los argumentos de las dos operaciones anteriores son siempre positivos.

(b) Modificar la respuesta del apartado anterior, de tal forma que el reintegro de fondos a los clientes sea servido según un orden FIFO. Por ejemplo, suponer que el saldo es 200 unidades y un cliente está esperando un reintegro de 300 unidades. Si llega otro cliente, debe esperarse incluso si quiere retirar 200 unidades. Suponer que existe una función denominada `cantidad(cond)` que devuelve el valor de la cantidad (parámetro c de los procedimientos retirar y depositar), que espera retirar el primer proceso que se bloqueó (tras ejecutar `cond.wait()`).

19. Considerar el programa concurrente mostrado más abajo. En dicho programa hay 2 procesos, denominados `P1` y `P2`, que intentan alternarse en el acceso al monitor `M`. La intención del programador al escribir este programa era que el proceso `P1` esperase bloqueado en la cola de la señal `p`, hasta que el proceso `P2` llamase al procedimiento `M.sigue()` para desbloquear al proceso `P1`; después `P2` se esperaría hasta que `P1` terminase de ejecutar `M.stop()`, tras realizar algún procesamiento se ejecutaría `q.signal()` para desbloquear a `P2`. Sin embargo el programa se bloquea.

(a) Encontrar un escenario en el que se bloquee el programa.

(b) Modificar el programa para que su comportamiento sea el deseado y se eviten interbloqueos.

```
Monitor M ( ) {
cond p, q;
procedure stop {
    begin
       p.wait();
       .......
       q.signal();
    end;
}
procedure sigue {
begin .......
   p.signal();
   q.wait();
    end;}
begin
end;
}
```

```
Proceso P1;
begin
 while TRUE do
   ...
   M.stop();
    ...
 end;
```

```
Proceso P2;
begin
 while TRUE do
       ...
     M.sigue();
      ...
 end;
```

20. Indicar con qué tipo (o tipos) de señales de los monitores (SC,SW o SU) sería correcto el código de los procedimientos de los siguientes monitores que intentan implementar un semáforo FIFO. Modificar el código de los procedimientos de tal forma que pudieran ser correctos con cualquiera de los tipos de señales anteriormente mencionados.

```
Monitor Semaforo{
 int s;
 cond c;
void* P(void* arg){
   if (s == 0) c.wait();
   s= s-1;
}
void* V(void* arg){
   s= s+1;
   c.signal();
}
begin
   s= 0;
end;
}
```

```
Monitor Semaforo{
 int s;
 cond c;
void* P(void* arg){
   while (s == 0){
      c.wait();
      }
      s= s - 1;
   }
void* V(void* arg){
   notifyAll();
   s = s+1;
}
begin
   s= 0;
end;
}
```

21. Suponer que "n" procesos comparten "m" impresoras. Antes de utilizar una impresora, el proceso P_i llama a la operación `int pedir( int id_proceso)`, que devuelve el número de impresora libre. En caso de que no existiera ninguna libre en ese momento, la llamada al método anterior ocasiona que el proceso P_i se quede esperando 1 impresora. Los

procesos después de utilizar una impresora llaman al método `public void devolver(int id_impresora)`. Cuando quede una impresora libre (la deja de utilizar el proceso que la estaba usando), si hay varios procesos esperando, se le concederá dicha impresora al proceso de mayor prioridad de los que esperan. La prioridad de un proceso viene dada por su índice de proceso. De tal forma que un proceso que tenga un índice menor tendrá mayor prioridad. Si cuando se deja libre una impresora no existiera ningún proceso esperando, entonces la impresora cuyo identificador aparece en la llamada al método `public void devolver(int id_impresora)` quedaría como impresora libre, es decir, no asignada todavía a ningún proceso. Se pide programar los métodos anteriormente citados dentro de 1 monitor suponiendo semántica de señales desplazantes y SU. Escribir también el invariante del monitor programado.

22. Escribir el código de los procedimientos `P()` y `V()` de un monitor que implemente un semáforo general con el siguiente invariante: $\{\mathtt{s} \geq 0\} \vee \{\mathtt{s} = \mathtt{s}_0 + \mathtt{nV} - \mathtt{nP}\}$ y que sea correcto para cualquier semántica de señales. La implementación ha de asegurar que nunca se puede producir "robo de señal" por parte de las hebras que llamen a las operaciones del monitor semáforo anterior.

23. Suponer un número desconocido de procesos consumidores y productores de mensajes de una red de comunicaciones muy simple. Los mensajes se envían por los productores llamando a la operación `broadcast(int m)` (el mensaje se supone que es un entero), para enviar una copia del mensaje `m` a las hebras consumidoras que previamente hayan solicitado recibirlo, las cuales están bloqueadas esperando. Otra hebra productora no puede enviar el siguiente mensaje hasta que todas las hebras consumidoras no reciban el mensaje anteriormente envíado. Para recibir una copia de un mensaje enviado, las hebras consumidoras llaman a la operación `int fetch()`. Mientras un mensaje se esté transmitiendo por la red de comunicaciones, nuevas hebras consumidoras que soliciten recibirlo lo reciben inmediatamente sin esperar. La hebra productora, que envió el mensaje a la red, permanecerá bloqueada hasta que todas las hebras consumidoras solicitantes efectivamente lo hayan recibido. Se pide programar un monitor que incluya entre sus métodos las operaciones: `broadcast(int m)`, `int fetch()`, suponiendo una semántica de señales desplazantes y SU.

24. Suponer un sistema básico de asignación de páginas de memoria de un sistema operativo que proporciona 2 operaciones: `adquirir(int n)` y `liberar(int n)` para que los procesos de usuario puedan obtener las páginas que necesiten y, posteriormente, dejarlas libres para ser utilizadas por otros. Cuando los procesos llaman a la operación `adquirir(int n)`, si no hay memoria disponible para atenderla, la petición quedaría pendiente hasta que exista un número de páginas libres suficiente en memoria. Llamando a la operación `liberar(int n)` un proceso convierte en disponibles `n` páginas de la memoria del sistema. Suponemos que los procesos adquieren y devuelven páginas del mismo tamaño a un área de memoria con estructura de cola y en la que suponemos que no existe el problema conocido como fragmentación de páginas de la memoria. Se pide programar un monitor que incluya las operaciones anteriores suponiendo semántica de señales desplazantes y SU, así como que la que las llamadas a la operación para adquirir páginas se sirva en orden FIFO, es decir, aquellas hebras bloqueadas que representan a procesos de usuario que llamaron primero a la operación `adquirir(int n)` serán servidas antes.

25. Diseñar un controlador para un sistema de riegos que proporcione servicio cada 72 horas. Los usuarios del sistema de riegos obtienen servicio del mismo mientras un depósito de capacidad máxima igual a `C` litros tenga agua. Si un usuario llama a `abrir_cerrar_valvula (int cantidad)` y el depósito está vacío, entonces ha de señalarse al proceso controlador y la hebra que soporta la petición del citado usuario quedará bloqueada hasta que el depósito esté otra vez completamente lleno. Si el depósito no se encontrase lleno o contiene menos agua de la solicitada, entonces el riego se llevará a cabo con el agua que haya disponible en ese momento. La ejecución de la operación `control_en_espera()` mantiene bloqueado al controlador mientras el depósito no esté vacío. El llenado completo del depósito se produce cuando el controlador llama a la operación `control_rellenando()`, de tal forma que cuando el depósito esté completamente lleno (=`C` litros) se ha de señalar a las hebras–usuario bloqueadas para que terminen de ejecutar las operaciones de "abrir y cerrar válvula" que estaban interrumpidas. Se pide: programar las 3 operaciones mencionadas en un monitor que asumen semántica de señales desplazantes y SU. Demostrar que las hebras que llamen a las operaciones del monitor anterior nunca pueden llegar a entrar en una situación de bloqueo indefinido.

```
Proceso(i)::
 do{
   Riegos.abrir_cerrar_valvula(necesito);
   \\Regar ...

} while (true)
```

```
Controlador(i)::
 do{
 \\El deposito incialmente lleno
    \\Esperar 72 horas
   Riegos.control_en_espera();
   Riegos.control_rellenando();
} while (true)
```

Fuentes Consultadas

[Dahl et al., 1970] Dahl, O., Myhrhaug, B., and Nygaard, K. (1970). *SIMULA:Common Base Language*. Norwegian Computing Center, Oslo.

[Dijkstra, 1965] Dijkstra, E. (1965). Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569.

[Dijkstra, 1971] Dijkstra, E. (1971). Hierarchical ordering of sequential processes. *Acta Informatica*, 1:115:138.

[Hansen, 1977] Brinch-Hansen, P. (1977) The architecture of concurrent programs Prentice-Hall, Englewood Cliffs.

[Hoare, 1999] Hoare, C. (1999). Monitors: an operating system structuring concept. *Communications of the ACM*, 10:594–557.

[Holt, 1983] Holt, R. (1983). *Concurrent Euclid, The UNIX System and Tunis*. Addison-Wesley, Reading, Massachusetts.

[Holt et al., 1987] Holt, R., Matthews, P., Roselet, J., and Cordy, J. (1987). *The TURING programming language: design and definition*. Prentice-Hall, Englewood Cliffs.

[Lampson and Redell, 1980] Lampson, B. and Redell, D. (1980). Experience with processes and monitors in mesa. *Communications of the ACM*, 23(2):105–117.

[Lister, 1977] Lister, A. (1977). The problem of nested monitor calls. *Operating Systems Review*, 11(3):5–7.

[Peterson, 1983] Peterson, G. (1983). A new solution to lamport's concurrent programming problem using small shared variables. *Transactions on Programming Languages and Systems*, 5(1):56–65.

[Welsh and Bustard, 1979] Welsh, J. and Bustard, D. (1979). Pascal–plus: another language for modular multiprogramming. *Software Practice and Experience*, 9:947–957.

[Wirth, 1985] Wirth, N. (1985). *Programming in Modula-2*. Springer Verlag, Berlin.

Capítulo 3

Sistemas basados en paso de mensajes

3.1 Introducción

Los programas de un computador *Von Neumann* clásico se conciben para su ejecución determinista como un único programa secuencial. Sin embargo, el deseo de mayor velocidad de ejecución ha tenido como consecuencia la introducción del paralelismo en los programas y la aparición de los multiprocesadores y el multiprocesamiento.

Multiprocesador

Se trata de sistemas de computador que incluyen varias unidades de procesamiento (CPU), denominados genéricamente *procesadores*, que comparten la memoria principal y periféricos con el objetivo de ejecutar programas simultáneamente. Actualmente se habla de *multiprocesamiento simétrico*, también conocido como SMP según su acrónimo en inglés. Se trataría de una arquitectura hardware para un tipo de multiprocesador en la cual varios procesadores idénticos están conectados a una sola memoria principal compartida, abstrayendo el control individual de cada uno de ellos mediante una instancia del sistema operativo, que actúa de interfaz con los usuarios y programas de aplicación. La arquitectura SMP (ver figura 3.1) es hegemónica actualmente, especialmente para los denominados *procesadores multinúcleo* que la usan para tratar a los diferentes núcleos de un procesador como si fueran procesadores distintos. En la arquitectura SMP, los procesadores pueden ser interconectados mediante *buses*, *conmutadores de barras cruzadas* o redes de *tipo malla* dentro del propio chip. Ejemplos de algunos sistemas SMP, que merecen citarse por la difusión que han obtenido, son los siguientes: AMD(Athlon II, Opteron), Intel(CoreiX, Xeon, Itanium), Sun (UltraSPARC) y ARM (MPCore).

A fin de completar esta revisión, haremos referencia al término *procesamiento asimétrico* (AMP) (ver figura 3.1). Históricamente el modelo de multiprocesador AMP fue un recurso de software provisional para poder gestionar sistemas con procesadores de diferentes tipos antes de la aparición de los sistemas SMP. En los sistemas AMP la CPU no es más que el procesador aritmético y lógico que ejecuta las aplicaciones de usuario, no posee la funcionalidad de procesamiento gráfico, *multienhebramiento*, etc. de los procesadores multinúcleo actuales. Además, todas las CPUs de un multiprocesador AMP han de poseer el mismo conjunto de instrucciones de bajo nivel para las aplicaciones que las utilicen, ya que un trabajo en ejecución

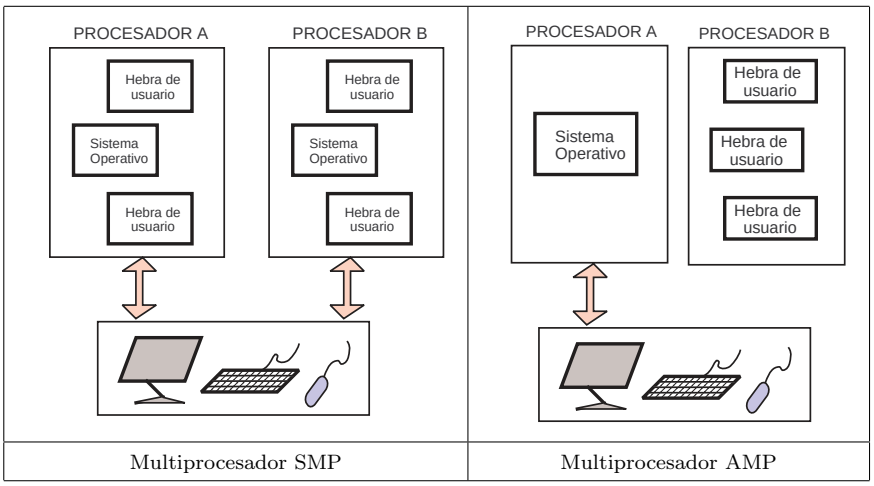


Figura 3.1: Tipos de multiprocesadores actuales

podría ser reasignado dinámicamente de una CPU a otra.

Los sistemas AMP presentaban un problema de programabilidad, ya que los sistemas operativos de la época se desarrollaban para una única CPU. No se llegó a resolver satisfactoriamente hasta la aparición de los sistemas SMP, en los que el sistema operativo y las aplicaciones bajo el control de este se ejecutan en todos los procesadores simultáneamente, con un tick de reloj de granularidad fina común. Ejemplos de sistemas AMP, que se utilizaron en la época anteriormente mencionada, son los siguientes: IBM(65MP), Burroughs B5000, CDC (6700), DEC (PDP-11 y VAX), Multics y UNIVAC (1108). En la actualidad los sistemas cuyas CPUs no son todas iguales suelen seguir otros modelos AMP, tales como multiprocesamiento de acceso a memoria no uniforme (NUMA) y multiprocesamiento en *racimos* (clustered).

Multiprocesamiento

Multiprocesamiento, o *multiproceso*, consiste en la utilización de dos o más CPUs en un mismo sistema multiprocesador para ejecutar los programas de una misma aplicación. El término *multiprocesamiento* también se refiere a la habilidad de un sistema de este tipo para gestionar más de un procesador y/o poder reasignar tareas entre dichos procesadores durante la ejecución de los programas mencionados.

El término multiprocesamiento se utiliza a veces para referirse a la ejecución de múltiples procesos software concurrentes en un sistema, en contraposición con la ejecución de un único proceso en cualquier instante de la ejecución de un programa, típico de los sistemas monoprocesador. Sin embargo, tal concepto resulta más apropiado indicarlo con cualquiera de los términos *multiprogramación* o *multitarea*, ya que estos suelen implicar una implementación en software, mientras que *multiprocesamiento* es un término más apropiado para describir la ejecución de un programa por múltiples procesadores–hardware. Un sistema puede realizar ambas cosas: multiprocesamiento y multiprogramación, o sólo una de éstas. Actualmente son muy raros los sistemas de computador que sólo tengan un procesador mononúcleo y que, por tanto, trabajen sin presentar ningún tipo de multiprocesamiento o multiprogramación.

Desde el punto de vista del modelo de ejecución de instrucciones de un programa por parte de un multiprocesador, los procesadores pueden utilizarse para una única secuencia o varias secuencias de instrucciones que se ejecutan en contextos múltiples. Flynn propuso una

clasificación (ver tabla 3.1) de los tipos de multiprocesamiento que se ha mantenido hasta la fecha.

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

Tabla 3.1: Taxonomía del multiprocesamiento.

El modelo `SIMD` describe a los multiprocesadores que ejecutan una única secuencia de instrucciones en diferentes contextos de ejecución. Es decir, todos los procesadores podrían sincronizarse para ejecutar la misma instrucción simultáneamente, pero que el resultado afectase a datos ubicados en distintas localizaciones de la memoria. El tipo SIMD está indicado para el procesamiento paralelo de vectores, en los cuales una gran cantidad de datos pueden ser divididos en partes que son modificadas independientemente. En este modelo una única secuencia de instrucciones dirige la operación de múltiples CPUs para que realicen las mismas manipulaciones sobre los datos, incluso sobre grandes cantidades diferentes a la vez.

Los multiprocesadores del modelo `MISD`, por el contrario, ejecutarían instrucciones distintas, que pertenecen posiblemente a diferentes secuencias de ejecución de los procesos de un programa, pero que afectan a la misma zona de memoria. El modelo `MISD` principalmente ofrece la ventaja de la redundancia en los cálculos, ya que múltiples procesos realizan las mismas tareas sobre los mismos datos, reduciendo las posibilidades de que se produzcan resultados incorrectos si una de éstas fallase. Las arquitecturas `MISD` pueden conllevar el realizar comparaciones entre diferentes procesadores para detectar fallos.

Aparte de las características de redundancia y seguridad de este tipo de multiprocesamiento, el modelo `MISD` posee muy pocas ventajas y resulta muy caro de implementar. No mejora el rendimiento de los programas. Puede ser implementado de tal manera que la existencia de los diferentes procesadores resulte transparente al software. Un ejemplo de la utilidad del modelo `MISD` es el proceso progresivo de imágenes, donde cada pixel de una imagen es *encauzado* a través de varios procesadores hardware que realizan pasos de transformación de dicha imagen.

En el modelo `MIMD` el procesamiento de las secuencias de instrucciones de los programas se divide en múltiples hebras, cada una de las cuales posee su propio estado del procesador dentro de uno o múltiples procesos definidos por software. Dado que ahora la mayoría de los procesadores son multinúcleo y trabajan con múltiples hebras que esperan ser planificadas, ya se trate de hebras del sistema o de los programas de usuario, este modelo de arquitectura está recomendado para hacer buen uso de los recursos de bajo nivel de los procesadores actuales.

El uso de multiprocesadores que siguen el modelo `MIMD` puede suscitar problemas de contención de recursos por los procesos e incluso conducir a interbloqueos, ya que las hebras pueden entrar en conflicto, de una forma impredecible, en su acceso a los recursos, que resulta difícil de prever y gestionar eficientemente.

Implementación del modelo de ejecución

La implementación del modelo `MIMD` necesita una codificación especial a nivel del sistema operativo, pero no requiere cambios sustanciales en las aplicaciones desarrolladas para un sistema monoprocesador, salvo que los programas de usuario utilicen múltiples hebras. Es decir, el modelo `MIMD` es transparente para los programas con una única hebra de control si el programa

no cede voluntariamente dicho control al sistema operativo durante su ejecución [1].

Por otra parte, con multiprocesadores que siguen el modelo `MIMD`, tanto el software del sistema como los programas de usuario pueden necesitar utilizar construcciones software, tales como semáforos o *cerrojos*, para impedir que una hebra interfiera a otra si se da el caso en que ambas entrelacen sus ejecuciones mientras referencian los mismos datos. La necesidad de acceso exclusivo a determinados datos durante la ejecución de las hebras asíncronas incrementa la complejidad del código, hace bajar el rendimiento de los programas, y convierte en imprescindible la verificación del código, aunque no hasta el punto de anular las ventajas del multiprocesamiento. Conflictos similares suelen aparecer entre los procesadores a nivel del hardware, por ejemplo, situaciones de contención y corrupción de datos en el cache. Normalmente dichos problemas deben ser resueltos a nivel de hardware, o mediante una combinación de software y hardware utilizando instrucciones para borrar el cache en los programas.

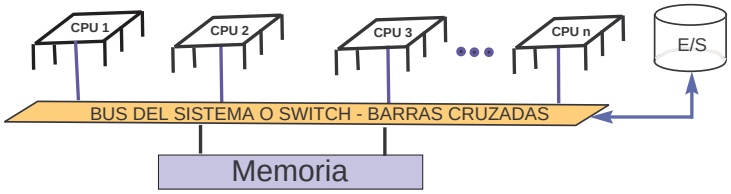


Figura 3.2: Representación de un multiprocesador con memoria compartida

Hay varias posibilidades de implementación del modelo MIMD:

1. Utilizar variables en memoria común a los procesadores (ver figura 3.2)
   - En este tipo de multiprocesador la sincronización entre los procesos está basada en la secuencialización que se produce al intentar acceder varios procesos a una misma dirección de memoria.
   - Es típica en este modelo un tipo de programación concurrente basado en monitores, semáforos, regiones críticas, etc.
   - Se necesitan dispositivos especiales para que los procesadores puedan acceder con eficiencia a posiciones de memoria compartida.
   - El inconveniente principal de estas arquitecturas es la falta de escalabilidad.
2. No existe memoria común, toda la comunicación y sincronización ha de llevarse a cabo a través de una red de comunicaciones (ver figura 3.3). Por eso se les llama *multicomputadores*.
   - Son más difíciles de programar que los multiprocesadores de memoria común, sin embargo presentan otras ventajas que los hacen ser las máquinas actualmente más utilizadas.
   - No presentan el problema de la escalabilidad de los multiprocesadores de memoria común.
   - Se necesita una notación de programación lo más flexible posible que permita expresar los diferentes modos de comunicación entre los procesos, así como expresar el no-determinismo en las comunicaciones.

[1] por ejemplo, provocando una llamada al sistema operativo

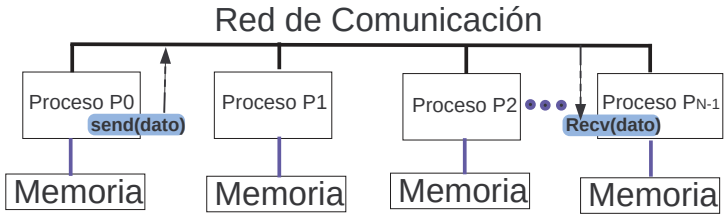


Figura 3.3: Representación de un multicomputador

- Las primitivas concurrentes clásicas: semáforos, regiones críticas, monitores, etc. no son adecuadas para programar los multicomputadores.

Estructura SPMD de un programa de paso de mensajes

Ejecutar un programa diferente en cada proceso, según el modelo general de la figura 3.3, puede ser algo difícil de manejar. Se suele utilizar un estilo de programación distribuida denominado *Single Program Multiple Data* (SPMD) en el cual el código que ejecutan los procesos es idéntico, pero sobre datos diferentes (ver figura 3.4). El estilo SPMD es una variante del modelo general `MIMD` que se aproxima al `SIMD`, en tanto en cuanto los procesadores ejecutan un mismo programa, pero no tienen que sincronizarse en la ejecución de cada una de las instrucciones individuales como ocurriría en un multiprocesador `SIMD`.

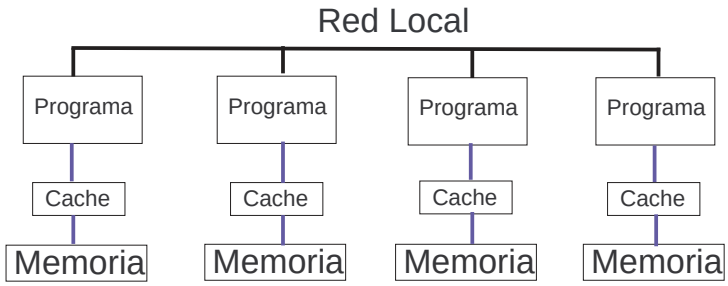


Figura 3.4: Representación del modelo de programación SPMD

Cada programa incluye la lógica interna necesaria para que cada proceso ejecute la tarea que le corresponda de acuerdo con el valor de su identificador, que será distinto para cada uno de ellos, y suele establecerse en una etapa de configuración distribuida posterior a la compilación.

3.2 Mecanismos básicos en sistemas basados en paso de mensajes

En los sistemas multicomputador, los procesos de un programa concurrente se comunican y sincronizan mediante *paso de mensajes*. Por *comunicación* entendemos que los procesos se envían y reciben mensajes en lugar de acceder en escritura o lectura a las variables compartidas del programa. La *sincronización* entre procesos se produce como consecuencia de que la recepción de un mensaje es posterior al envío del mismo y, de forma general, la ejecución de la operación de recibir supone la espera de la disponibilidad del mensaje en la parte receptora.

Las primitivas básicas de paso de mensajes se refieren al envío y recepción de una serie de datos entre el emisor del mensaje y el receptor:

- `send(<lista de variables>,<identificador_destino>)`
- `receive(<lista de variables>,<identificador_origen>)`

Para programar correctamente ha de tenerse en cuenta qué esquema concreto de *identificación de la comunicación* utiliza el lenguaje de programación o el sistema, así como el *modo de sincronización* o semántica de las primitivas de paso de mensajes anteriores.

Esquemas de identificación de los procesos comunicantes

Se trata de determinar cómo se identifican mutuamente los procesos emisores y receptores de mensajes durante la ejecución de un programa distribuido. En el caso de la llamada *denominación directa*, se utilizan los identificadores de los procesos para que el emisor pueda señalar explícitamente al receptor y viceversa. El problema principal que presenta esta opción consiste en que los identificadores de los procesos se han de asignar previamente a la ejecución del programa y dicha asignación se ha de mantener durante ésta. Como consecuencia de esto, cualquier cambio en la identificación de los procesos requiere recompilar el código. Este tipo de denominación tiene como ventaja principal que no produce ningún retardo debido a la identificación, aunque sólo permite comunicaciones uno–a–uno entre los procesos.

```
Proceso P0
int dato;
Produce(dato);
send(&dato,P1);
```

```
Proceso P1
int x;
receive(&x,P0);
Consume(x);
```

El esquema de identificación entre procesos que presenta una mayor flexibilidad, ya que permite varias configuraciones de comunicación entre grupos de procesos, se llama *denominación indirecta*. Se basa en la utilización de un objeto intermedio denominado *buzón* entre los procesos comunicantes, de esta forma los procesos designan al buzón como destino u origen de los mensajes que van a intercambiarse y, por tanto, se evita la restricción de los enlaces uno–a–uno que implica la utilización del esquema de denominación directa. Existen tres tipos de buzones, dependiendo de la relación que se establezca entre los procesos.

Relación	1-a-1	muchos-a-1	muchos-a-muchos
Tipo	Canales	Puertos	Buzones generales

Tabla 3.2: Tipos de buzones

El destino de los mensajes que se envían a través de un puerto es un único nodo, pero el origen del mensaje puede ser uno entre un conjunto y no es necesario especificarlo. Por tanto, los puertos pueden servir para multiplexar múltiples puntos de destino en un único nodo de la red. Cada proceso en los programas recibirá la información remota a través de sus puertos, lo cual permite la utilización de más de un servicio de red simultáneamente. Los puertos son parte de la capa de transporte en el modelo TCP/IP y de la capa de sesión en el modelo ISO.

En el caso de los buzones generales el destino de los mensajes envíados por un proceso puede ser cualquier nodo de la red. Así como también, cualquier nodo puede ser origen de un

mensaje recibido en el destino. Se pueden entender como puertos que multiplexan tanto los puntos de destino como los de origen en un único nodo de la red. Como consecuencia de ello tienen una implementación más complicada e ineficiente que los puertos si no existe una red de comunicaciones especializada. En general, el envío de un mensaje implica su transmisión al resto de la red y la recepción significa notificar la disponibilidad del mensaje a todos.

```
Channel of Integer Buzon;
  Proceso P0                          Proceso P1
  int dato;                           int x;
  Produce(dato);                      receive(&x,Buzon);
  send(&dato,Buzon);                  Consume(x);
```

Los canales se pueden entender como un tipo especial de puerto que sólo tiene un nodo origen. También como un servicio de comunicación orientado a establecer conexiones entre los procesos de las aplicaciones software, similar al concepto de *circuito virtual* entre nodos de una red. Con los canales se puede transmitir un flujo de datos simple, de tal forma que dichos datos son entregados en el orden en que se enviaron, sin que su división en paquetes o marcos ocasione que se desordenen durante la transmisión por la red. Los canales se pueden implementar utilizando Transmission Control Protocol (TCP), mediante un protocolo específico que proporciona circuitos virtuales encima del protocolo IP[2], que no está orientado a establecer conexiones confiables entre nodos y, por tanto, no garantiza que se mantenga el orden de envío de los datos en la recepción. El protocolo X.25 proporciona comunicación nodo-a-nodo confiable y garantiza al mismo tiempo la calidad de servicio en las comunicaciones; este protocolo proporciona identificadores de canales virtuales (VCI) en las aplicaciones.

Desde el punto de vista de la implementación, si dos procesos interactuantes se ubican en un mismo procesador, entonces el medio de transmisión de los mensajes puede ser simplemente la memoria local del procesador. Si, por el contrario, estos están en diferentes procesadores, entonces el paso de mensajes entre los dos procesos ha de ser realizado a través de un medio de comunicaciones físico que conecte a ambos. Actualmente, el paso de mensajes ha de ser entendido como una primitiva de comunicación y de sincronización entre procesos más cercana a los lenguajes de programación que a la plataforma [Andrews, 1991][3].

Semántica de las operaciones de paso de mensajes

El significado o *semántica* de estas operaciones puede ser diferente, es decir, el resultado de su ejecución podría variar dependiendo de la *seguridad* y el *modo de comunicación* que necesite un programa o aplicación. Por tanto, existen diferentes versiones de las operaciones de paso de mensajes que garantizan o no la seguridad en la transmisión de los datos y diferentes modos de comunicación.

La propiedad de *seguridad* en el paso de mensajes se cumple por parte de la operación *send* cuando la ejecución de esta operación garantiza que el valor recibido por el proceso destino sea el que tenían los datos justo antes de la llamada. En el ejemplo siguiente se considera que la operación *send* en el proceso P_0 posee semántica segura y que el proceso P_1 siempre imprimirá el valor `100` cada vez que se ejecute dicho código:

[2]el circuito virtual es establecido identificando el par de direcciones de los *sockets* de red receptor y emisor, es decir, sus direcciones IP y números de puerto

[3]plataforma de computación = sistema operativo + nivel de red (según la normalización ISO)

```
Proceso P0
int dato=100;
send(&dato,P1);
```

```
Proceso P1
int x;
receive(&x,P0);
imprime(x);
```

Una operación de envío con semántica *insegura* podría ocasionar que el valor recibido por P_1 fuese distinto de `100` si, por ejemplo, el valor de `dato` es alterado después de volver la llamada a *send* pero antes de que el sistema comience a transmitir el valor de la variable.

Por otra parte, la llamada a la operación *receive* no necesariamente detiene al proceso receptor P_1 y, por tanto, el valor de la variable `x` podría ser alterado por otras instrucciones antes de que termine la transmisión física de `dato`. Normalmente, los sistemas que ofrecen una operación *receive* no bloqueante también poseen una operación de comprobación que indica en qué momento se pueden alterar los datos que son recibidos y de esta forma se pueda programar con una operación *receive* semánticamente segura.

3.2.1 Operaciones bloqueantes

La semántica de este tipo de operaciones de paso de mensajes implica que la llamada a la operación *send* sólo vuelve cuando se garantice la propiedad de seguridad (ver sección 3.2). No siempre ocurriá que el proceso receptor haya recibido el dato cuando termine la ejecución de *send*, sino más bien que los cambios que se hayan producido en los datos durante la comunicación no violarán la citada semántica de estas operaciones.

Modo comunicación	Hardware especializado	Sincronización	Seguridad
Sin búfer	-	Sí (citas)	Sí
Con búfer	Sí	Relajada	Sí
	No	Sí	Sí

Tabla 3.3: Características de las operaciones bloqueantes

Paso de mensajes síncrono (sin búfer)

En el paso de mensajes síncrono la comunicación se lleva a cabo mediante un enlace directo entre los procesos participantes. Supongamos que un proceso A le envía datos a un proceso B. Cuando el proceso A ejecute la operación *send* se esperará hasta que el proceso B ejecute la operación de recepción *receive*. Antes que los datos se puedan transmitir físicamente los procesos han de encontrarse preparados para participar en el intercambio, lo cual exige una *cita* entre el emisor y el receptor (ver figura 3.5). Es decir, de forma similar a lo que ocurre en el emisor, la llamada a *receive* en el proceso receptor se suspende hasta que el otro proceso llame a la operación *send*.

El concepto de *cita* entre procesos comunicantes implica:

- Sincronización entre emisor y receptor para que se produzca el intercambio.
- El proceso emisor podrá realizar aserciones acerca del estado del receptor en el punto de sincronización.

- Puede considerarse un tipo de comunicación análoga a una conversación telefónica o un *chat* de dos participantes.

Las operaciones bloqueantes proporcionan a los programas de aplicación un tipo de comunicación que respeta la semántica de seguridad en el paso de mensajes en todos los casos, sin embargo suele tener una implementación ineficiente en los sistemas si los procesos participantes en una *cita* no están preparados para al mismo tiempo. En la figura 3.5 se puede ver que los procesos emisor o receptor sufren espera ociosa si el otro participante no ha llegado todavía a ejecutar su operación de recepción o envío del mensaje, respectivamente.

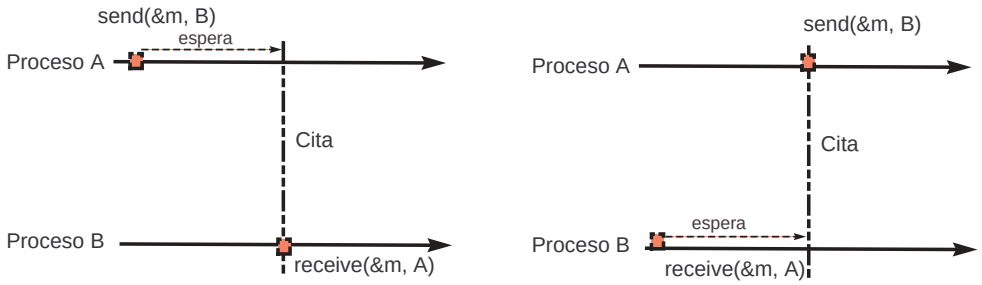


Figura 3.5: Cita entre procesos con operaciones de comunicación síncronas

Otro problema que afecta a las propiedades de seguridad sería la posibilidad de que aparezcan interbloqueos si no se intercambian las operaciones de envío y recepción que se refieran al mismo mensaje. El siguiente código producirá el bloqueo indefinido de ambos procesos si las operaciones *send* y *receive* son bloqueantes:

```
Proceso P0
send(&dato_1,P1);
receive(&x_2,P1);
```

```
Proceso P1
send(&dato_2,P0);
receive(&x_1,P0);
```

Paso de mensajes con búfer

En este tipo de paso de mensajes, la operación *receive* posee la misma semántica (significado/comportamiento) que en el paso de mensajes síncrono. Sin embargo, la primitiva de paso de mensajes `send()` posee una semántica diferente. El medio de comunicación entre los procesos no es ahora un enlace directo entre los 2 procesos que participan en la comunicación, sino más bien una cola de mensajes. La existencia de un hardware de comunicación especializado permite al proceso A continuar con su ejecución después de llamar a la operación *send*, ya que cuando el proceso A envía al proceso B, el mensaje se añade al búfer que representa la cola de mensajes pendientes de ser recibidos. Para recibir un mensaje, el proceso B ejecuta la operación *receive* que elimina el mensaje situado en la cabeza de la cola de mensajes, continuando después con su ejecución. Si no hay mensajes que recibir, es decir, el búfer se encuentra vacío, entonces la primitiva de comunicación `receive()` bloqueará al proceso receptor hasta que algún proceso emisor provoque que el búfer deje de estar vacío.

Implementaciones de bajo nivel

Existen dos variantes, interesantes de comentar, al modelo básico de paso de mensajes bloqueante. La primera se aplica a los sistemas que utilizan *canales* (ver sección 3.2) como medio

de comunicación. Consiste en que algunos sistemas implementan una primitiva denominada operación *vacío*, que testea el contenido de un determinado canal, y devuelve el valor `true` si no hay mensajes. La utilidad de esto sería la de prevenir el bloqueo de la llamada a la operación *receive* cuando, en ausencia de mensajes presentes en el canal, se puede aprovechar el tiempo realizando algún trabajo útil alternativo a la espera ociosa.

La segunda variante consiste en que la mayoría de los sistemas basados en paso de mensajes bloqueante utilizan un *búfer* interno[4] de longitud fija en el receptor (ver figura 3.6). Cuando este ejecuta la operación *receive*, el sistema comprueba si el mensaje está ya disponible en el búfer y, si lo está, copia los datos en el área de memoria donde el proceso receptor espera recibir el mensaje. Si la plataforma donde se ejecuta cuenta con hardware especializado, la transferencia de los datos se iniciará inmediatamente después de que sean copiados en el búfer interno, sin que se vea interrumpido el proceso receptor, por tanto, decimos que la sincronización entre los procesos emisor y receptor se ve relajada en este caso (ver tabla 3.3). Por el contrario, si no se cuenta con un hardware de comunicaciones específico, el proceso emisor interrumpirá al receptor, interviniendo ambos procesos en la transferencia interna de datos al búfer del receptor. Posteriormente, cuando el receptor llama a la operación *receive*, se copia el mensaje aludido desde el búfer a la zona de memoria asignada para recibirlo.

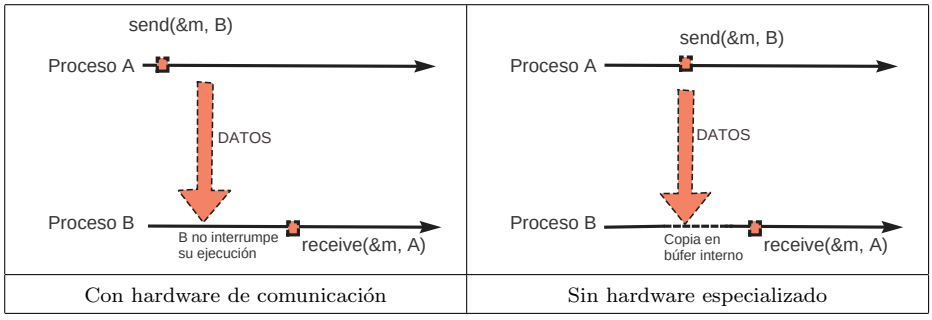


Figura 3.6: Implementación del paso de mensajes bloqueante con búfer

3.2.2 Operaciones no-bloqueantes

Las operaciones bloqueantes garantizan comunicaciones con semántica segura respecto de los datos que se transmiten, pero adolecen de ineficencia a la hora de su implementación en plataformas que no posean un hardware de comunicaciones especializado:

Paso de mensajes	Motivo de la ineficiencia
Síncrono	Espera ociosa
Con búfer	Sobrecarga por gestión del búfer

Tabla 3.4: Problemas de implementación del paso de mensajes bloqueante

Por consiguiente, en lugar de utilizar este tipo de operaciones se podría pensar en definir operaciones *send* y *receive* que no bloquean y dejar en la responsabilidad del programador el

[4]no confundir con la estructura de datos programada por el usuario en programas que siguen el modelo productor-consumidor

asegurar la semántica en el paso de mensajes que programe sus aplicaciones. Esto se puede lograr permitiendo que las operaciones aludidas devuelvan el control al programa donde se ejecutan antes incluso de que sea seguro modificar los datos. El programador se encargará de asegurar que no se alteren los datos de los programas mientras están siendo transmitidos si esto puede ocasionar errores en las aplicaciones. Para poder llevarlo a cabo, han de existir *sentencias de comprobación de estado* que indican si en un momento dado se pueden alterar los datos sin provocar que la semántica deje de ser segura. De esta forma, una vez iniciada la operación de paso de mensajes, el programa podría realizar cualquier cálculo que no dependa de la finalización de la operación y comprobará la terminación de ésta cuando sea necesario.

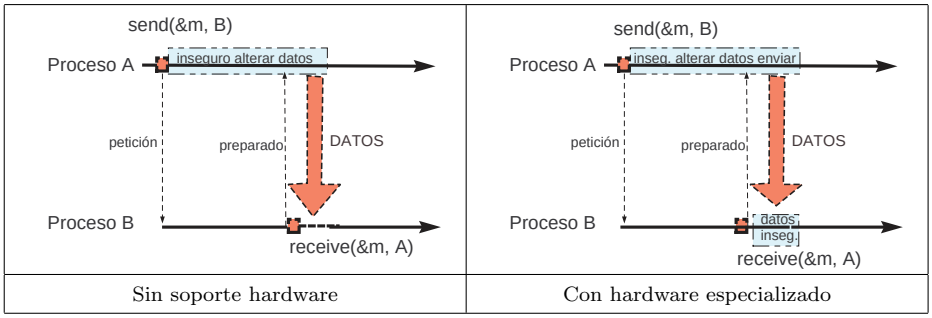


Figura 3.7: Paso de mensajes no-bloqueante sin búfer

Paso de mensajes sin búfer

La ejecución de una operación *send* con operaciones no bloqueantes informa al sistema que hay un mensaje pendiente, pero el proceso emisor continua su ejecución después de haberlo enviado. De esta manera se pueden iniciar otros cálculos, no necesariamente relacionados con la comunicación, por parte del programa mientras el mensaje se encuentra en transmisión. Cuando en el otro proceso se confirme la llamada a la operación *receive* del mensaje, se iniciará la comunicación física entre el emisor y el receptor. Si no existe soporte de hardware especializado, el proceso receptor se suspende desde que el sistema está preparado para recibir los datos hasta el final de la transmisión. Con soporte de hardware especializado, la operación *receive* vuelve inmediatemente, aunque no se hayan terminado de transmitir los datos del mensaje. Existe una operación de comprobación que indicará cuándo es seguro acceder a los datos que están siendo comunicados (ver figura 3.7).

Paso de mensajes con búfer

La diferencia fundamental con el modo *sin búfer* consiste en que cuando se llama a la operación *receive* se inicia la transferencia de los datos del mensaje desde el búfer interno al área de memoria del receptor donde espera hallarlos. Como consecuencia, se reduce el tiempo de espera en el receptor durante el cual un acceso a dichos datos es inseguro.

3.2.3 Tipos de procesos en programas de paso de mensajes

1. **Filtros**: son procesos transformadores de datos. Reciben flujos de datos de sus canales de entrada, realizan algún cálculo en los flujos de datos, y envían los resultados a los canales de salida.
2. **Clientes**: son procesos desencadenantes de algo. Los clientes hacen peticiones a los

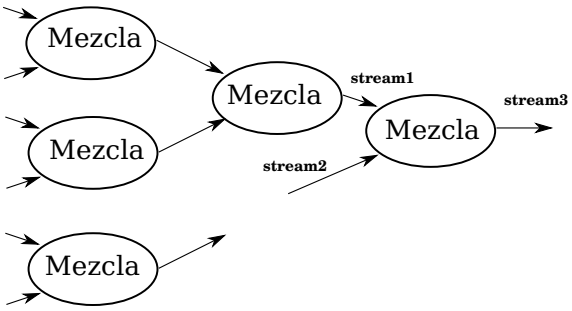


Figura 3.8: Procesos de tipo filtro.

procesos servidores y desencadenan reacciones en los servidores. Los clientes inician una actividad, cuando ellos eligen el momento, y a menudo se esperan hasta que se haya servido la petición.

3. **Servidores**: son procesos reactivos. Esperan hasta que se hagan las peticiones y, entonces, reaccionan a una petición. La acción específica que toman depende de la petición, los parámetros de la petición y el estado del servidor. El servidor puede responder inmediatamente o puede tener que guardar la petición para responderla después. Un servidor es un proceso que nunca termina y que a menudo sirve a más de un cliente.

```
const EOS = high (int); \\ fin del marcador de flujo
op stream1 (x:int), stream2 (x:int), stream3 (x:int);
process mezcla{
  int  v1, v2;

  receive stream1 (v1);
  receive stream2 (v2);

  do (v1 < EOS and v2 < EOS)
    if (v1 <= v2)
       send stream3 (v1);
       receive stream1 (v1);
    [] v2 <= v1 ->
       send stream3 (v2);
       receive stream2 (v2);
    fi
  od;

  if (v1 = EOS)
    send stream3 (v2);
  [] (v1 <> EOS) ->
    send stream3 (v1);
  fi;

  send stream3 (EOS)
}
```

Figura 3.9: Implementación de los procesos filtro “mezcla”

4. **Pares**: son procesos idénticos que interaccionan para proporcionar un servicio o resolver un problema.

3.3 Modelos y lenguajes de programación distribuida

En lo que sigue nos centraremos en los multicomputadores y en los modelos y lenguajes de programación adecuados para estas plataformas de computación.

Para poder programar procesos que sean del tipo procesos *servidores*, los lenguajes introducen una nueva sentencia de programación denominada *orden guardada*. La semántica de esta orden proviene de la idea de considerar la selección entre varias alternativas de forma *no–determinista* como una *ayuda mental* [Dijkstra, 1975], más que como un inconveniente, en el desarrollo de programas distribuidos. Hasta el artículo aludido, los informáticos pensaban en las estructuras no-determinísticas como *algo a eliminar* en los programas, ya que se consideraban como una fuente de posibles errores en la fase de mantenimiento posterior del software. La práctica tradicional era eliminarlo total o parcialmente en la fase de codificación, intentándose preveer qué alternativa en concreto tomaría un programa en el momento de su ejecución. Sin embargo, contar con sentencias no-deterministas en los lenguajes de programación puede tener sentido para facilitar la implementación de un determinado tipo de sistemas que reaccionan frente estímulos procedentes de su entorno.

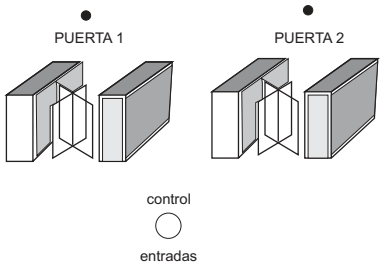


Figura 3.10: Modelo hardware de control de entrada a un museo

Considérese, por ejemplo, el caso de la implementación de un controlador para contar la entrada a un museo con dos puertas (ver figura 3.10). Los sensores de detección de presencia de cualquiera de las puertas pueden enviar al controlador una señal de que ha entrado una nueva persona; pero no se puede predecir el orden de envío de las señales, de hecho podrían recibirse al mismo tiempo.

```
P1::                        P2::                        P3::\\Proceso controlador
  for(i=1;i<20;i++){        for(i=1;i<20;i++){          for (j=1;j<20;j++){
      send(P3,1);              send(p3,1);                 receive(P1, temp);
      --pasa 1 persona         --pasa 1 persona            cont+=temp;
  }                           }                           receive(P2, temp);
                                                          cont+=temp;
                                                       }
                                                       printf("%d", cont);

main(){
   cobegin P1;P2;P3 coend;
}
```

Figura 3.11: Implementación inadecuada de un controlador para detectar entradas

Una implementación totalmente determinista de controlador, como la que se muestra en la figura 3.11, es totalmente errónea. Si suponemos paso de mensajes de tipo síncrono, el proceso

P3, que representa al controlador, podría bloquearse si no se producen entradas desde la puerta 1, aunque entren personas por la puerta 2. El proceso P3 que implementa al controlador es un proceso de tipo *servidor* y, por tanto, no conoce de antemano al proceso cliente que se va a comunicar con él en cada instante.

3.3.1 Espera selectiva con órdenes guardadas

Un *servidor* es un tipo de proceso que ha de estar preparado para recibir un mensaje de cualquiera de sus clientes, sin que el orden en que se produzcan dichas comunicaciones pueda ser determinado a priori. Además, dependiendo del estado de los datos en el servidor, en cada iteración de dicho proceso podrían estar permitidas las comunicaciones de sólo algunos clientes. Pensar por ejemplo en un proceso servidor que implementa una cola circular (*búfer*) con las operaciones clásicas de *inserción* para los productores y *eliminación* para los consumidores. Cuando se llene el búfer de datos, no se recibirá más de los productores hasta que al menos un consumidor establezca la comunicación y elimine un dato. Análogamente, cuando el búfer se quede vacío, no se aceptarán comunicaciones con los consumidores hasta después de recibir al menos un dato de un productor. Cuando el búfer se encuentre en un estado distinto de los dos anteriores, se podrán aceptar comunicaciones de los productores y consumidores en un orden *no determinado*.

Construcción	Propósito
Orden guardada	Permitir una comunicación condicional con un cliente
Espera selectiva	no determinismo en selección de alternativas *guardadas*

Tabla 3.5: Órdenes estructuradas para lenguajes con operaciones de comunicación síncronas

Órdenes guardadas

Dijkstra propuso unas nuevas construcciones, para ser incluidas en los lenguajes de programación distribuidos y con operaciones de comunicación síncronas, que llamó *ordenes guardadas*. Las mencionadas órdenes pasan a ser las sentencias componentes básicas de la construcción denominada *espera selectiva* de un lenguaje del tipo anterior:

```
   <espera.selectiva>::= SELECT <conjunto.ordenes.guardadas> END SELECT
<conjunto.ordenes.guardadas>::= <orden.guardada>OR <orden.guardada>}
          <orden.guardada>::= <guarda> -> <lista.sentencias>
                    <guarda>::= <expresion.booleana> |
                    <expresion.booleana>;receive(<argumentos>);|
                     receive(<argumentos>)
```

Se dice que una orden guardada está *preparada* para ser ejecutada si la expresión booleana (o condición) que precede a la instrucción *receive* se evalúa como cierta y la propia operación *receive* está lista para recibir el mensaje.

Es el propio proceso servidor que programa la orden de espera selectiva quien selecciona en cada llamada a la orden SELECT, de una forma totalmente incontrolable desde los procesos

que constituyen su entorno, qué orden guardada concreta va a ejecutarse entre las incluidas en el subconjunto de las *preparadas*. La orden puede incluirse dentro de un bucle, de tal forma que en cada iteración se ejecutará una nueva instancia de la orden SELECT.

```
PUERTA(i:1..2)::
{ int s=0;
  do
   SELECT (s<HORA.CIERRE && PERSONA())->
       {send(CONTROL, S()); // envia una se\~nal de entrada de persona
        DELAY.UNTIL(s+1); // espera hasta el siguiente instante
        s:=s+1;} // cuenta un nuevo tick de reloj
     OR
       (s<HORA.CIERRE && NOT PERSONA())->
       {DELAY.UNTIL(s+1); // espera hasta el siguiente instante
        s:=s+1;} // cuenta otro tick
     OR
       TRUE->DELAY.UNTIL (TIME() + 16*3600); // es la hora de
       // cierre del museo; hay que esperar 16 horas.
       // TIME() devuelve una cuenta en segundos.
  END SELECT
  while(true);
  send(CONTROL, Start();
}
CONTROL::
{ int cont= 0;
  SELECT  receive(Start(), PUERTA(1)); // desde cualquiera de los sensores
      OR                               // de las puertas se arranca
         receive(Start(), PUERTA(2)); // el controlador
  END SELECT
  do
     SELECT receive(S(), PUERTA(1))-> cont:= cont+1;
         OR
            receive(S(), PUERTA(2))-> cont:= cont+1;
     END SELECT
     // cuenta una persona mas, porque ha recibido la se\~nal
     // de cualquiera de las 2 puertas (no se puede saber cual)
  while(true);
  printf("numero de personas",%d, cont));
}
main(){
  cobegin PUERTA;CONTROL coend;
}
```

Figura 3.12: Implementación del controlador con órdenes con guarda

Resumen de la semática de la orden SELECT:

- El subconjunto de órdenes guardadas *preparadas* se determina una sola vez al comienzo de su ejecución.
- Para volver a determinar qué órdenes están preparadas hay que ejecutar nuevamente la orden.
- Mientras un proceso cliente no realice algún envío que empareje con alguna de las órdenes guardadas cuya condición se evaluó como cierta, la orden producirá un bloqueo ya que no posee ninguna orden guardada ejecutable en ese momento.
- Algunos lenguajes[5] permiten programar SELECT con alternativas prioritarias, pero en estos casos la selección dejaría de ser no determinista.

Por tanto, las órdenes guardadas permiten implementar selecciones no-deterministas en los programas, es decir, la alternativa realizada e incluso el estado final no dependen únicamente del estado inicial que tuviera el proceso antes de ejecutar la orden con guarda seleccionada.

En la figura 3.12 se puede ver una implementación correcta, con órdenes guardadas y espera selectiva, inspirado en un lenguaje distribuido [Hoare, 1985], para el ejemplo del controlador para la entrada de personas.

3.4 Bibliotecas de paso de mensajes

Actualmente MPI (Message Passing Interface) se ha convertido en un estándar, siendo una interfaz muy útil para la realización de aplicaciones paralelas basadas en paso de mensajes [Snir et al., 1999]. El modelo de programación paralela y distribuida que se puede realizar con MPI es el denominado MIMD, aunque se suele utilizar bastante con un modelo que es un caso particular, el denominado modelo SPMD (ver tabla 3.1). En el modelo SPMD todos los procesos ejecutan el mismo programa, aunque no necesariamente la misma instrucción al mismo tiempo. MPI es, como su nombre indica, un interfaz, lo que quiere decir que el estándar no exige una determinada implementación del mismo. Lo importante es dar al programador una colección de funciones para que este diseñe su aplicación, sin que tenga necesariamente que conocer el hardware concreto sobre el que se va a ejecutar, ni la forma en la que se han implementado las funciones que emplea.

El desarrollo de la interfaz MPI y sus implementaciones se ha debido a MPI Forum, un grupo formado por investigadores de universidades, laboratorios y empresas involucrados en la computación paralela, también denominada de *altas prestaciones* (HPPC, según su acrónimo en inglés). Básicamente MPI pretende definir un único entorno de programación, que garantice la total portabilidad de las aplicaciones paralelas, basado en una única interfaz y sin especificar cómo se debe llevar a cabo la implementación de ninguna de ellas. También ofrece a los usuarios y programadores implementaciones, de dominio público, de dicho entorno que aseguran un nivel de calidad con el objetivo de propiciar la difusión del estándar.

[5] Ada 95, por ejemplo, permite definir alternativas de mayor prioridad en la selección no determinista

Los elementos básicos de MPI son una definición de un interfaz de programación independiente del lenguaje, más una colección de implementaciones de ese interfaz (o *bindings*, como se dice sucintamente) para los lenguajes de programación más extendidos en la comunidad usuaria de computadores paralelos, es decir: Ada, C y FORTRAN. Cualquiera que quiera utilizar MPI para desarrollar software ha de trabajar con una implementación de MPI que constará de, al menos, los siguientes elementos:

- Una biblioteca de funciones para C, más el archivo de cabecera `mpi.h` con las definiciones de esas funciones y de una colección de constantes y macros.
- Una biblioteca de funciones para FORTRAN, junto con el archivo de cabecera `mpif.h`.
- Comandos para compilación, típicamente `mpicc, mpif77`, que son versiones de las órdenes de compilación habituales (`cc, f77`), que incorporan automáticamente las bibliotecas MPI.
- Órdenes específicas para la ejecución de aplicaciones paralelas, normalmente denominada `mpirun`.
- Herramientas para monitorización y depuración de programas paralelos.

MPI no es, evidentemente, el único entorno disponible para la elaboración de aplicaciones paralelas, ya que existen otras muchas alternativas, tales como:

- Utilizar las bibliotecas de programación propiedad del computador paralelo disponible:
  - NX en el Intel Paragon
  - MPL en el IBM SP2, etc.
- PVM (Parallel Virtual Machine), que posee unas características similares a MPI e intenta hacer que una red de estaciones de trabajo funcione como un multicomputador.
- Lenguajes de programación paralelos o que incluyan directivas de paralelismo.
- Lenguajes de programación secuenciales, junto con compiladores que paralelicen automáticamente el código producido por los programas.

Como anteriormente se ha comentado, MPI está diseñado pensando en el desarrollo de aplicaciones SPMD. Este tipo de programas lanzan en paralelo N copias de un mismo programa, que se ejecutan por procesos asíncronos, es decir, hay que programar en el código de los procesos las instrucciones necesarias de semáforos para sincronizarlos. Dado que los procesos en MPI poseen un espacio de memoria completamente separado, el intercambio de información, así como la sincronización entre ellos, se ha de hacer exclusivamente mediante paso de mensajes. MPI ofrece a los programadores tanto operaciones de paso de mensajes bloqueantes como no bloqueantes. Las primeras facilitan una programación más fácil y segura, mientras que las segundas permiten optimizar el rendimiento de los programas distribuidos al enmascarar las sobrecargas debidas a la comunicación y transmisión de datos. En MPI se dispone de operaciones punto-a-punto para 2 procesos comunicantes, así como funciones u operaciones colectivas para involucrar a un grupo de procesos. Los procesos pueden agruparse y formar

comunicadores, lo que permite una definición del ámbito de las operaciones colectivas, así como propician el desarrollo de un diseño modular de las aplicaciones.

A continuación se ve un ejemplo de programa que utiliza el *binding* de MPI para el lenguaje de programación C:

```
# include "mpi.h"
#include <iostream>
using namespace std;
main (int argc, char **argv) {
  int nproc; /*Numero de procesos */
  int yo; /* Mi direccion: 0<=yo<=(nproc-1)*/
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
  MPI_Comm_rank(MPI_COMM_WORLD, &yo);
  /* CUERPO DEL PROGRAMA */
  cout<<"Soy el proceso " <<yo<<" de "<<nproc<<endl;
  MPI_Finalize();
}
```

En el código anterior se nos presentan 4 de las funciones más utilizadas de MPI: `MPI_Init`() para iniciar la ejecución paralela, `MPI_Comm_size`() para determinar el número de procesos que participan en la aplicación, `MPI_Comm_rank`(), para que cada proceso obtenga su identificador dentro de la colección de procesos que componen la aplicación, y `MPI_Finalize`() para terminar la ejecución del programa.

Si utilizamos MPI, entonces los nombres de todas las funciones han de comenzar con `MPI_`, la primera letra que sigue siempre es mayúscula, y el resto son minúsculas. La mayoría de las funciones MPI devuelven un entero, que hay que interpretarlo como un diagnóstico. Si el valor devuelto es `MPI_SUCCESS`, la función se ha realizado con éxito. La palabra clave `MPI_COMM_WORLD` se refiere al comunicador universal, es decir, un comunicador predefinido por MPI que incluye a todos los procesos de nuestro programa. Se pueden definir otros comunicadores en MPI y todas las funciones de comunicación de MPI necesitan como argumento un comunicador.

3.5 Mecanismos de alto nivel en sistemas distribuidos

En el modelo de paso de mensajes caracterizado por comunicación síncrona y paso de mensajes unidireccional, cada canal de comunicación es utilizado para pasar información en una dirección solamente, entre un único proceso emisor y un único proceso receptor. Con estas primitivas de paso de mensajes de bajo nivel se puede implementar cualquier tipo de interacción entre procesos. Pero no se adaptan a todos los esquemas comunicación. Por ejemplo, para el esquema cliente-servidor se tiene una implementación demasiado forzada.

```
proceso cliente                    proceso servidor
 do                                      do
   TRUE ->                                 (i:0..n) condicion(i);
     send(servidor,peticion());             receive (cliente[i], peticion()) ->
     receive(servidor,respuesta);            realizar.servicio();
 od                                          send(cliente(i),resultado);
                                         od
```

La solución anterior produce un código poco seguro, sobre todo si las peticiones de servicio de los clientes han de aguardar un mensaje por parte del servidor que indique que se ha completado. De forma similar, el servidor puede verse afectado por un cliente no fiable que falle antes de recibir el segundo mensaje (`receive(servidor,respuesta)`). De darse esto se produciría el bloqueo del proceso servidor.

El par `(send(servidor,peticion()), receive(servidor,respuesta))` ha de ser una única transacción lógica y no es adecuado representarla como 2 operaciones de paso de mensajes síncronos independientes. Por lo tanto, el canal de comunicación entre el proceso cliente y el servidor debería soportar comunicación en los 2 sentidos.

3.5.1 El modelo de *llamadas remotas*

Es un modelo de comunicación de paso de mensajes síncrono, pero que tiene muchas de las características de las *llamadas a procedimiento* de los lenguajes secuenciales:

- Permite implementar de una forma flexible los procesos con una relación tipo cliente-servidor.
- Se permite a varios procesos llamar concurrentemente a un procedimiento que es poseído y controlado por otro proceso (comunicación muchos-a-1).
- Una llamada remota a procedimiento puede implicar paso de información en 2 direcciones.
- El procedimiento llamado encapsula una serie de instrucciones que son ejecutadas en nombre del proceso llamador (cliente), antes de que se devuelva ningún resultado.
- Este modelo admite varias implementaciones; de las cuales se van a estudiar: *llamada a procedimiento remoto* (RPC) y la *invocación remota* (o modelo basado en citas).
- *remoto* para las RPC's significa en un procesador distinto y *remoto* para las invocaciones remotas significa en un proceso distinto.

3.5.2 Llamada a Procedimiento Remoto *RPC*

Concepto de Procedimiento Remoto
Es un mecanismo que permite a un programa, ejecutándose en un nodo de una red, ejecutar un procedimiento en otra máquina. Las llamadas a procedimientos o métodos remotos tienen una sintaxis similar a la llamada a un procedimiento dentro de un programa secuencial, pero una

semántica completamente diferente. Ha de existir un programa ejecutántose en un procesador remoto (servidor) para ejecutar las llamadas. Podemos pensar en el procedimiento remoto como un *procedimiento global* que es llamado por los procesos clientes. El procedimiento es ejecutado por un proceso creado a tal efecto en el servidor que ejecuta su cuerpo y devuelve un mensaje con los resultados.

Algunas implementaciones de este modelo se han demostrado particularmente útiles y eficientes en sistemas distribuidos. Dicho modelo ha sido adoptado como mecanismo de comunicación a bajo nivel en versiones distribuidas del sistema operativo UNIX.

Una descripción general de la semántica de la *llamada a procedimiento remoto* es la siguiente:

- Se envían los argumentos de la llamada al servidor, bien directamente, o través de un proceso creado a tal efecto (*stub*).
- Se bloquea el proceso cliente que ejecuta la llamada. Este proceso puede dejar su procesador libre mientras se gestiona la llamada remota.
- Se reconstruyen los argumentos de la llamada en el servidor, se ejecuta el procedimiento y se reenvían los argumentos resultado al proceso cliente.
- El servidor suele ligar un nombre simbólico al puerto del que recibe las llamadas para ejecutar un procedimiento remoto determinado.
- Se pueden tener varias instancias de un mismo procedimiento ejecutándose concurrentemente, si cada llamada de un proceso cliente crea un nuevo proceso en el servidor. En este caso, las variables del procedimiento han de poder ser accedidas concurrentemente, por lo tanto hay que asegurar la exclusión mutua en dicho acceso.

3.5.3 Implementación del modelo en Java: RMI

La tecnología estándar que propone Java [Lea, 2001] para programación distribuida se denomina *Remote Method Invocation (RMI)*. Dicha tecnología está basada en el modelo de llamadas remotas y en la programación orientada a objetos, de ahí que se hable de invocación de métodos remotos.

Utilizando RMI y el lenguaje de programación Java en la parte del servidor se ha de programar una clase que implemente los métodos que van a ser llamados remotamente. La información concreta acerca de qué métodos de una clase pueden ser llamados por otros procesos no se encuentra dentro de la clase, sino que se ha de incluir en una sub-interfaz de `Remote`:

```
public interface Hello extends Remote{
  public String sayHello() throws java.rmi.RemoteException;
}
```

La sub-interfaz *Remote* permite, en el nivel de implementación, para cada servidor RMI, registrar los objetos disponibles y los nombres de los métodos que pueden ser llamados remotamente.

```
public class HelloImpl extends UnicastRemoteObject implements Hello {

  public HelloImpl() throws RemoteException{
    super();
  }
  public String sayHello() throws RemoteException{
    returns ‘‘Hello World!!!’’;
  }
  public static void main(String args[]){
    try{
      HelloImpl h = HelloImpl();
      Naming.rebind(‘‘hello’’, h);
      System.out.println(‘‘Hello server ready.’’);
    }
    catch(RemoteException re){
      ...
    }
    catch (MalformedURLException e){
      ...
    }
  }
}
```

Se ha de declarar un método constructor en la clase que contiene al método remoto, ya que la cláusula que especifica tirar la excepción es obligatoria.

En el método `main()` se liga el objeto devuelto por el constructor al nombre simbólico ‘‘`hello`’’. Esta asociación entre un nombre simbólico y un objeto es incluida en el registro de nombres del servidor y hecha accesible a futuros procesos clientes.

Los procesos clientes, para poder utilizar el servicio del método `sayHello()`, han de programar en su código:

```
public static void main (String args[]){
  System.setSecurityManager(new RMISecurityManager());

  try{
    Hello h = (Hello) Naming.lookup(‘‘rmi://ockham.ugr.es/hello’’);
    String message = h.sayHello();
    System.out.println(‘‘HelloClient:’’ + message);
  }
  catch(remoteException re){
   ...
  }
}
```

Según se puede ver representado en la figura 3.13, antes de que el servidor pueda comenzar a aceptar llamadas de los procesos clientes, se han de generar los *stubs* y los *skeletons*. Cada

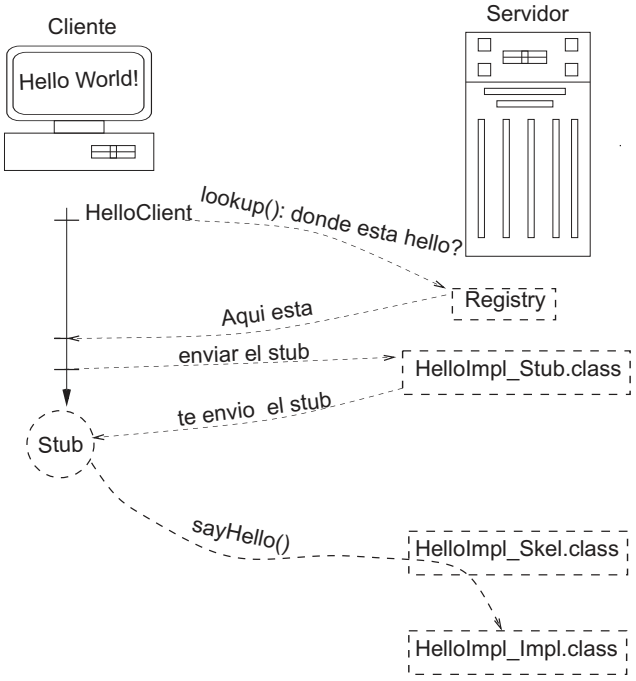


Figura 3.13: Representación de la ejecución de `HelloImpl.class`

stub contiene la signatura[6] de los métodos incluidos en la interfaz remota. Un *skeleton* tiene una función similar al *stub*, pero en la parte del servidor. Los *stubs* y *skeletons* son generados automáticamente en el servidor, a partir del código fuente de la clase (`HelloImpl` del ejemplo anterior).

Posteriormente habría que iniciar la ejecución del servidor de nombres (registry) y después lanzar el programa en el servidor (`java HelloImpl&`).

3.5.4 Semántica del paso de parámetros en RMI

Java, en las llamadas a métodos, pasa por referencia los parámetros referidos a objetos locales (no *por copia*). Si se mantuviera el mismo sistema de paso de parámetros para las llamadas a métodos remotos, el lenguaje sería poco útil para desarrollar programas distribuidos, ya que sería bastante ineficiente pasar por referencia objetos que contienen sólo datos, tales como: arrays, registros o cadenas. Si, por ejemplo, se pasa *por referencia* un array de 100,000 elementos a un método remoto, para procesar completamente dicho array se necesitaría realizar 100,000 accesos a una máquina remota.

Para evitar la disfunción que produciría disponer en Java sólo de *paso por referencia* en las llamadas a los métodos, cuando se programa con RMI, se establecen las siguientes reglas de paso de parámetros para métodos remotos:

- los argumentos de una llamada que sean de un tipo primitivo se pasarán *por copia*,
- los parámetros referidos a objetos se pasarán por referencia, o no, dependiendo de las interfaces que implementen las clases a las que pertenecen dichos métodos:

[6] nombre, lista y tipo de parámetros de una función o método

```
public interfaca PilaRemota extends
 Serializable{
   public int[ ] volcar()
      throws RemoteException;
        return celdas;
 }
 public class RMI_test{
  static PilaRemota s1 = null;
  static PilaRemota s2 = null;
   public static void main(String[ ] args){
  //objetos-pila creados en diferentes sitios
     s1.insertar(3);
     s2.insertar (3);
     int[ ] pila1 = s1.volcar();
     int[ ] pila2 = s2.volcar();
     pila1[0] = 100;
     pila2[0] = 100;
     System.out.println("pila1: "+s1.cima());
     System.out.println("pila2: "+s2.cima());
   }
}
```

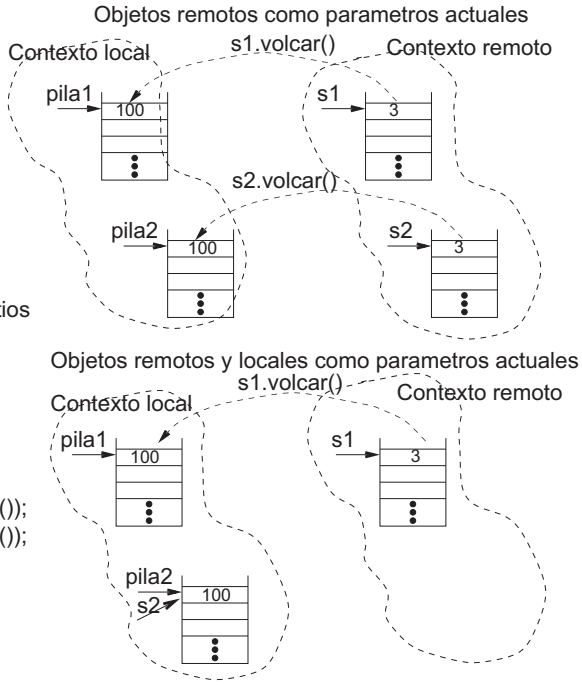


Figura 3.14: Paso de parámetros por referencia y copia en Java

- *remote interface:* se pasan por referencia,
- *serializable interface:* se pasan por copia.

Java permite la coexistencia de 2 semánticas distintas para la llamada a los métodos de una misma variable-objeto. La sintaxis para invocar a los métodos es la misma, independientemente de que el método pertenezca a la interfaz de un objeto local o remoto respecto del que lo llama, sin embargo, la ejecución de un método puede producir diferentes resultados, según sea local o remoto, según se muestra en la figura 3.14.

Puesto que Java permite el polimorfismo cuando se programa con RMI, no hay forma de averiguar de manera estática[7] si un objeto es local o remoto cuando se invocan sus métodos. Como consecuencia de ello, suele ser complicado programar aplicaciones distribuidas correctas, con la semántica del paso de parámetros en las invocaciones a métodos remotos de la definición actual del lenguaje Java.

3.5.5 Implementación del modelo con *invocación remota*

Son sentencias de un lenguaje de Programación Concurrente que permiten a un proceso llamar a un procedimiento que pertenece y es controlado por otro proceso. En este caso, el procedimiento remoto es una secuencia de una ó más instrucciones que pueden estar situadas en cualquier parte del código de un proceso.

A diferencia de las RPC's el proceso que llama y el proceso al cual pertenece el procedimiento remoto pueden ejecutarse por el mismo procesador.

[7]por ejemplo, siguiendo todas las referencias a los objetos de un programa.

Cada invocación remota se implementa definiendo un *punto de entrada* y una o varias *sentencias de aceptación* asociadas a dicha entrada. Ada [Barnes, 1994] es un lenguaje de programación que sigue este modelo y ha alcanzado mayor difusión hasta la fecha.

Definicion de los *puntos de entrada*

En un proceso se define un punto de entrada por cada uno de los procedimientos que pueden ser llamados desde otros procesos. Las entradas son definidas en el contexto del proceso que las posee: un proceso que hace una llamada tiene que indicar el punto de entrada y el proceso que la posee. A diferencia de las RPC's, las entradas no son nombres globales del sistema, por lo tanto, pueden estar repetidas en diferentes procesos.

Mientras que un canal podía tener como máximo un proceso esperando comunicación, los puntos de entrada pueden tener cualquier número de llamadas pendientes. Los datos se comunican entre los procesos a través de los parámetros declarados en los puntos de entrada; se sigue una notación similar a la declaración de parámetros en los procedimientos:

Los procesos clientes llaman a los puntos de entrada; ya no es necesario declarar canales entre cada cliente y el proceso servidor.

Por cada punto de entrada existe una cola que atiende las llamadas de los procesos según el orden de llegada (cola FIFO) para contener las llamadas pendientes de los procesos clientes.

```
PROCESO BUFFER
  ENTRY   depositar(dato:  tipo.dato);
  ENTRY   tomar(VAR x:tipo.dato);
```

No se especifica ningún orden de declaración de los procedimientos remotos; los procesos pueden hacer llamadas de sus puntos de entrada que presenten circularidad:

```
PROCESO.A          PROCESO.B
  ENTRY E;           ENTRY F;
BEGIN              BEGIN
  B.F;               A.E
END                END
```

Comunicación mediante citas

El código correspondiente a la invocación remota se ejecutará cuando el proceso que tiene definida la entrada lo permita, a diferencia del procedimiento llamado durante una RPC, que es ejecutado inmediatamente, puesto que se trata de una *llamada a procedimiento* similar a las de los lenguajes secuenciales.

Por cada punto de entrada, en el códido de un proceso, hay que declarar al menos una sentencia de aceptación que contiene el código a ejecutar cuando se produzca la llamada al

punto de entrada. La ejecución de la sentencia de aceptación bloquea hasta que se produce la llamada de otra tarea al punto de entrada (si existieran varias llamadas esperando en la cola del punto de entrada, se atendería la primera).

Por lo tanto, la sentencia de aceptación se ejecuta sólo cuando las dos procesos están preparados para la comunicación:

```
Proceso P                        Proceso P'
 ENTRY E(...)                      ....
 ...                               BEGIN
 BEGIN                               P.E(...);
   ACCEPT E(...);                    ...
   ...                              END;
 END;
```

La comunicación que se establece entre el proceso `P` y el proceso `P'` se denomina *cita* o *rendez-vous*, dicho mecanismo pertenece a un modelo síncrono de comunicación.

Antes y después de producirse la cita los dos procesos se ejecutan asíncronamente.

La sintaxis de la sentencia de aceptación varía si se permite compartir variables globales entre procesos concurrentes, en este caso, se ha de añadir un cuerpo a dicha sentencia, esto es, una zona de código que se ejecuta en exclusión mutua.

3.6 Problemas resueltos

Ejercicio 1

Simular un semáforo general con citas.

Solución:

```
wait()                          signal()                          proceso
------                          --------                          -------
ENTRY semaforo_wait(int S);     ENTRY semaforo_signal(int S)      semaforo_signal(S)
BEGIN                           BEGIN                             ...
  accept_semaforo_wait(S);        accept_semaforo_signal(S);      semaforo_wait(S)
    when s>0                      do
    do                              S++;
        S--;                      end do;
    end do:                     END
END
```

Ejercicio 2

Simular una cita mediante semáforos.

Solución:

```
inic(S1,0)
inic(S2,0)
inic(mutex,1)


P1                  P2
--                  --
wait(S1);           signal(S1)
wait(mutex);        wait(S2)
//cita
signal(mutex);
signal(S2);
```

Ejercicio 3

Un tren tiene N asientos libres. Un pasajero sube al tren y ocupa su plaza, tras lo cual quedaría 1 asiento libre menos. Si todas las plazas del tren se ocupan o si durante 30 minutos no toma asiento ningún nuevo pasajero, entonces el tren realiza 1 viaje, tras lo cual vuelven a quedar N asientos libres. Suponer que para realizar 1 viaje, el tren ha de tener ocupado al menos 1 asiento; ya que si el tren tuviera todos sus asientos libres, se quedaría esperando indefinidamente a que se subiera 1 pasajero y ocupara su asiento. Suponer las siguientes condiciones para poder programar la actuación del tren y de los pasajeros con *citas*: 1) hay un número ilimitado de pasajeros y cada uno realiza 1 solo viaje, Suponer que 1 proceso pasajero cuando consigue asiento termina; 2) el tren está realizando viajes continuamente. Se pide resolver el problema en los 2 casos siguientes:

- **Se supone 1 único tren, por lo tanto se puede programar su actuación como un solo proceso servidor.**
- **Ahora se suponen 2 trenes. En este caso si 1 pasajero intenta subir al tren y no puede, esperará 1 minuto, tras lo cual, lo intentará con el segundo tren; si tampoco pudiera subir al segundo, lo volvería a intentar con el primer tren transcurrido 1 minuto y así sucesivamente.**

Solución:

```
Tren
----
ENTRY llega();
libres=0;
do{
    SELECT
        when libres>0 ->
            accept llega(); libres--;
    or
        when libres == 0;
            delay 0;
            // realizar viaje
            sleep(random()%MAX);
            libres=N;
    or
        when libres < N ->
            delay 30*60; // espera 30 min
            // realizar viaje
            libres=N;
    END SELECT
}while(true);
```

```
Pasajero (caso 1)
------------
do{
    Tren.llega();
}while(true);

Pasajero (caso 2)
------------
do{
    SELECT
        Tren1.llega(); return;
    or
        delay 1*60; // espera
    END SELECT
    SELECT
        Tren2.llega(); return;
        delay 1*60;
    END SELECT
}while(true);
```

Ejercicio 4

Se tienen N procesos cliente que interactúan con el proceso servidor de 1 cajero automático. Los procesos cliente obtienen dinero del cajero realizando la siguiente operación:

- **Informan de su identidad al cajero y solicitan una cantidad de dinero.**
- **El cajero responde con la cantidad solicitada si el cliente tiene suficiente saldo, si no, el cajero responde denegando la petición al cliente.**
- **Para ingresar dinero en el cajero los procesos sólo tienen que identificarse e indicar la cantidad ingresada.**
- **Suponer que cada cliente tiene inicialmente 10 unidades de saldo y que el cajero posee 100 unidades de efectivo para servir las peticiones de los clientes.**

Cuando el cajero agota completamente las 100 unidades de efectivo, no podrá servir peticiones de ningún tipo hasta pasada 1 hora, transcurrido ese tiempo se vuelven a reponer las 100 unidades de efectivo. Los ingresos de los clientes no incrementan las unidades de efectivo que tiene el cajero.

Solución:

```
ENTRY sacar(num_cliente: in int; cantidad:in/out int));
ENTRY ingresa(num_cliente: in int; cantidad:in/out int));
int efectivo=100;
int saldo[num_cliente];
do{
    SELECT
        when (efectivo>0) ->
            accept sacar(num_cliente,cantidad) do
                if (ifectivo>=cantidad and saldo[num_cliente>=cantidad]){
                    efectivo-=cantidad;
                    saldo[num_cliente]-=cantidad;
                } else cantidad=-1;
            end do;
    or
        when (efectivo>0)
            accept ingresar(num_cliente,cantidad)do
                saldo[num_cliente]+=cantidad;
            end do;
    or
        when (efectivo==0)
            delay 60*60;
            efectivo=100;
    END SELECT
}while(true);
```

Ejercicio 5

Programar una versión distribuida del problema de la cena de los 5 filósofos, suponiendo que se dispone de 5 procesos *filósofo*, 5 procesos *tenedores* cuyo comportamiento es el de un semáforo y un proceso habitación que limita la entrada a un máximo de 4 filósofos para que no se llegue a la situación de interbloqueo (cada uno de los filósofos ha cogido 1 tenedor y ninguno puede adquirir nunca más el que le falta para poder comer de la fuente de los espaguetis). El comportamiento de un proceso filósofo es un ciclo indefinido que repite: *pensar, entrar en la habitación, coger su tenedor izquierdo, coger su tenedor derecho, comer, soltar tenedor izquierdo, soltar tenedor derecho, salir de la habitación.*

Solución:

```
Process Tenedor(i)
------------------
do{
    receive(Filosofo(i),coger);
    receive(Filosofo(i),dejar);
}while(true)

Process Filosofo(i)
-------------------
do{
    send(Habitacion, entrar);
    send(Tenedor(i), coger); send(Tenedor((i+1)%5), coger);
    send(Tenedor(i), dejar); send(Tenedor((i+1)%5), dejar);
    send(Habitacion, salir);
}while(true)

Process Habitacion
------------------
do{
    SELECT
        when ocupado<4;
            receive(Filosofo(i), entrar);
            ocupado++;
        or
            receive(Filosofo(i),salir);
            ocupado--;
    END SELECT;
}while(true)
```

Ejercicio 6

Suponer que en un centro de proceso de datos hay 2 impresoras `A` y `B` que son similares, pero no idénticas. Tres clases de procesos clientes utilizan las impresoras: aquellos que necesitan utilizar la impresora `A`, los que necesitan utilizar la impresora `B` y aquellos otros que pueden utilizar cualquiera de las 2 impresoras: `A` o `B`. Cada tipo de proceso cliente ejecuta una llamada de petición de impresora y otra llamada para liberarla. Programar con citas un proceso servidor para asignar las impresoras anteriores. Se supone que 1 proceso cliente no puede quedarse con una impresora para siempre.
Solución:

```
Clientes A                    Clientes tipo=A|B
----------                    -----------------
do{                           do{
    servidor.cedeA();             servidor.cede(tipo);
    // usa impresora              // usa impresora
    servidor.libera(A);           servidor_libera(tipo);
}while(true);                 }while(true);

Servidor
--------
ENTRY cedeA();
ENTRY cedeB();
ENTRY cede(tipo:out tipo);
ENTRY libera(tipo: int tipo);
bool libreA=true;libreB=true;
do{
    SELECT
        when libreA ->
            accept cedeA();libreA=false;
    or
        when libreB ->
            accept cedeB();libreB=false;
    or
        when (libreA or libreB) ->
            accept cede (tipo: out tipo) do
                if(libreA){libreA=false;tipo=A;}
                else{libreB=false;tipo=B;}
    or
        accept libera(tipo: int tipo) do
            if tipo==A libreA=true;
            else libreB=true;
        end do;
    END SELECT
while(true);
```

1 - Si puede llegar a bloquearse, el proceso debe bloquearse para la copia de los datos al proceso que recibe. (NPI)

2 - MPI_Ssend() y MPI_Recv(). rico dice que tambien el wait()

3 - El problema es que el programa puede continuar sin la certeza de haber recibido/mandado los valores. Hace falta hacer 2 MPI_Wait() para estas dos operaciones, primero del send y luego del receive

4 - Simplemente espera al no haber ninguna guarda ejecutable.Ejecutará la de la primera llamada que reciba siempre que siga cumpliendo la condición.

5 - Salta una excepción.

3.7 Problemas propuestos

9. Son incompatibles ya que una clausula delay siempre se ejecutará cuando termine su tiempo de espera siempre que sea potencialmente ejecutable, por lo que no se activará la cláusula ELSE.

1. ¿Podría llegar a bloquearse temporalmente la orden de envío `send(...)` en el paso de mensajes asíncrono (*no bloqueante*) y buferizado? A qué se debería dicho bloqueo si es que crees que se puede producir.

2. ¿Qué operaciones de paso de mensajes de la biblioteca OpenMPI utilizarías para implementar el mecanismo de sincronización denominado *cita* entre 2 procesos?

3. ¿Qué problema tendría el siguiente código, que programa 2 operaciones de paso de mensajes *no bloqueante* y *no buferizado*, respecto de los valores finales de las variables `x, y`, declaradas en el programa y qué funciones de OpenMPI programarías en el espacio en blanco para resolver dicho problema?

```
int main(int argc, char *argv[]) { int rank, size, vecino, x, y;
MPI_Status status; MPI_Request request_send,request_recv;
MPI_Init(&argc, &argv); MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size( MPI_COMM_WORLD, &size ); y=rank*(rank+1); if (rank
mod 2 == 0) vecino=rank+1; else vecino=rank-1;
// Las siguientes operaciones pueden aparecer en cualquier orden
MPI_Irecv(&x,1,MPI_INT,vecino,0,MPI_COMM_WORLD,&request_recv);
MPI_Isend(&y,1,MPI_INT,vecino,0,MPI_COMM_WORLD,&request_send );
...
```

4. Explicar qué ocurre si en la ejecución de una orden de selección no-determinista todas las alternativas tienen condiciones evualadas a verdadero y una sentencia de aceptación (a continuación de cada condición), pero ningún otro proceso ha realizado todavía la llamada requerida por alguna de estas alternativas.

5. ¿Cuál sería el resultado de la ejecución de una selección no–determinista si todas las condiciones–guarda de sus alternativas se evalúan como falsas?

6. Considérense las siguientes tareas en un lenguaje distribuido con invocaciones remotas:

```
1 Process P1
2 do{
3 P2.A;
4 P3.B;
5 P4.G;
6 P3.E;
7 accept F;
8 P3.C;
9 }while(true);
10End P1;
```

```
1 Process P2
2 do{
3  select
4   accept A do
5     P3.D;
6   or
7   accept H do
8    P3.E;
9 end select;
10 while{true};
11End P2
```

```
1 Process P3
2 do{
3  accept D;
4  accept B;
5  accept E do
6  P1.F;
7 End P3;
8 accept C;
9 }while(true);
10End P3;
```

```
1 Process P4
2 do{
3  accept G;
4  T2.H;
5 }while(true);
6 End P4;
```

¿Puede producirse bloqueo global de los procesos anteriores? Si fuera así, indicar una secuencia de entrelazamiento que lleve a tal situación. Indicar el proceso y el número de línea de cada sentencia cuya ejecución provoque el bloqueo de cada proceso.

7. Programar la selección no–determinista del proceso `S1` a continuación para que se mantenga suspendida esperando la llamada de los otros procesos a su punto de entrada `E1()` durante 10.0 unidades de tiempo, transcurrido ese tiempo sin que se acepte la espera selectiva terminará.

```
Proceso S1(){                    Proceso S2(){
  ENTRY E1();                      ENTRY E2();


   SELECT                           SELECT
      ACCEPT E1();                     ACCEPT E2();



   END SELECT;                      END SELECT;
```

8. Programar la selección no–determinista del proceso `S2` para que si no hay ninguna llamada pendiente al punto de entrada `E2()`, por parte de algún cliente, la espera selectiva se cancele inmediatamente y, tras un retraso de 20.0 unidades de tiempo, la orden termine.

9. ¿Por qué en una orden de selección no determinista se pueden programar varias alternativas `delay <retraso>`, pero son incompatibles con la claúsula `ELSE`?

10. Dos tipos de personas, representados por los tipos de procesos `A` y `B` entran en 1 habitación. La habitación tiene una puerta muy estrecha por la que cabe 1 sola persona. La actuación de las personas es la siguiente:
    - Una persona de tipo `A` no puede abandonar la habitación hasta que encuentre a 10 personas de tipo `B`
    - 1 persona de tipo `B` no puede abandonar la habitación hasta que no encuentre a 1 persona d tipo `A` y otras 9 personas de tipo `B`.

    Siempre se ha de cumplir la siguiente condición: *en la habitación no hay personas de tipo* `A` *o hay menos de 10 personas de tipo* `B`. Si en algún momento no se cumple, tendrían que salir 1 persona de tipo `A` y 10 personas de tipo `B` para que se volviera a cumplir la condición. Cuando las personas salen de la habitación, no pueden entrar nuevas personas de ningún tipo hasta que salgan todos.

    Implementar una solución al problema anterior con citas. Para ello será necesario escribir un proceso servidor encargado de llevar a cabo la sincronización global de forma correcta. El código de los procesos cliente podría seguir el esqueleto mostrado más abajo. Las entradas `llamaA` y `llamaB` sirven para que los procesos cliente avisen al proceso servidor que desean sincronizarse de acuerdo con el esquema de sincronización indicado para resolver este problema. Las entradas `esperaA` y `esperaB` sirven para bloquear a los procesos clientes hasta que el número de clientes que han llamado a `llamaA` y `llamaB` sea el requerido para formar un grupo de personas de salida.

```
Process type tipoA {
   servidor.llamaA;
   servidor.esperaA;
   ...
}
```

```
Process type tipoB {
   ...
   servidor.llamaB;
   servidor.esperaB;
}
```

11. Terminar de programar la orden de selección siguiente, que implementa un semáforo binario de exclusión mutua ($0 \leq s \leq 1$), con sólo 2 puntos de entrada.

```
Proceso Mutex (int i){
 ENTRY wait();
 ENTRY post();
 int s=0;
  do{
    SELECT


    END SELECT;
 }while(true);
}
```

Con la implementación del semáforo anterior se pretende resolver el problema del acceso a la sección crítica por parte de dos procesos concurrentes. Para inicializarlo correctamente, 1 de los 2 procesos ha de ejecutar primero la llamada `Mutex(0).post()`:

```
void * p1( void * )
{
  //Solo lo inicializa 1 proceso
  Mutex(0).post();
  do {
     //Fuera de la seccion critica
     Mutex(0).wait();
     //Acceder a la
     //seccion critica
     //
     Mutex(0).post();
  }while(true);
  return NULL ;
}
```

```
void * p2( void * )
{
  do {
      //Fuera de la seccion critica
      Mutex(0).wait();
      //Acceder a la
      //seccion critica
      //
      Mutex(0).post();
  }while(true);
  return NULL ;
}
```

12. Programar el controlador de una máquina dispensadora de bebidas automática. Para cada bebida, acepta peticiones de los clientes: `ACCEPT peticion(num_bebida : IN int)`. Puede llegar a tener en total N latas de bebidas disponibles de los diferentes tipos. Cuando se agotan todas las latas entra en un periodo de mantenimiento durante 2 horas, tras el cual se reponen todas y la máquina vuelve a estar en servicio. Cada 24 horas se reponen las latas que faltan si se ha consumido alguna; esta operación tarda en realizarse 1/2 hora.
    - Completar el código de controlador de la máquina de bebidas, programando una selección no determinista, que se repite continuamente de acuerdo con las condiciones anteriores del problema:
    - Suponer ahora que hay 2 máquinas de bebidas en el edificio, con un funcionamiento totalmente equivalente. En este caso, si un cliente intenta obtener una bebida de 1 de las máquinas y no puede (la máquina está fuera de servicio), esperará 1 minuto, tras lo cual se desplazará al otro extremo del edificio para intentar sacar la bebida de la otra máquina. Si tampoco pudiera obtener una bebida, espera 1/2 hora a que la segunda máquina vuelva a estar en servicio. Si tampoco obtuviera la bebida con

la segunda máquina, lo volvería a intentar con la primera máquina, repitiendo las mismas acciones que realizó anteriormente, y así sucesivamente.

13. Conceptos generales sobre paso de mensajes. Responda Verdadero/Falso:

    (a) Los sistemas de paso de mensajes sólo pueden utilizarse en sistemas en los que no existe memoria compartida entre los procesos.

    (b) Las operaciones de paso de mensajes pueden ser utilizadas como primitivas de comunicación y sincronización, independientemente de si la plataforma es distribuida (*multicomputador*) o centralizada (*multiprocesador*).

    (c) Los procesos distribuidos y las hebras no son compatibles, por eso una aplicación que programe operaciones de paso de mensajes sólo admite que sus procesos estén ubicados en computadores diferentes.

    (d) Los canales de comunicación entre procesos distribuidos sólo pueden tener 1 proceso en cada extremo (*emisor* y *receptor*)

14. Responda Verdadero/Falso a las siguientes afirmaciones sobre los diferentes tipos de paso de mensajes:

    (a) Las ordenes de paso de mensajes sícronas nunca pueden llegar a bloquearse porque no se llenan búferes de mensajes que puedan desbordarse.

    (b) La operaciones de paso de mensaje síncrono (`Ssend()` y `Recv()`) nunca producen error si la sincronización de los procesos es correcta.

    (c) No hay forma de prevenir que las operaciones de recepción de mensajes síncrona, sin búfer, pueda bloquear a un proceso servidor si el proceso emisor ha terminado o ha abortado su ejecución.

    (d) En el caso de que un proceso intente enviar un valor a otro de forma asíncrona (no bloqueante y sin búfer) no debe utilizar los datos que va a enviar hasta que una función que ejecuta posteriormente le indique que puede hacerlo.

    (e) En el caso de paso de mensajes asíncrono el proceso receptor puede modificar los datos recibidos (`vector`) en cuanto vuelva la función `IRecv(&vector, N , ...)`.

15. Responda Verdadero/Falso a las siguientes afirmaciones sobre la sentencia `Select`:

    (a) Las condiciones–guardas de las alternativas sólo se evalúan 1 vez.

    (b) Si una condición se evalúa como falsa, se provoca el bloqueo indefinido del proceso que llama a la cita.

    (c) Si todas las condiciones de las alternativas regulares del `select` son falsas entonces se provoca un error de ejecución del programa.

    (d) Es conveniente programar siempre una claúsula `ELSE` en cada sentencia `Select` para evitar que se produzca el error si todas las alternativas están cerradas en 1 ejecución.

16. Responda Verdadero/Falso a las siguientes afirmaciones sobre la sentencia `Accept`:

    (a) El resultado del proceso P_1 a la orden de aceptación `ACCEPT f(IN x1, IN/OUT x2)` que ejecuta el proceso P_2 sólo tiene efecto en el proceso P_2, que luego envía dicho resultado al proceso P_1.

(b) Los procesos $\texttt{P}_1$ :: $\texttt{P}_2$.f(a, b) y $\texttt{P}_2$ :: ACCEPT f(IN $\texttt{x}_1$, IN/OUT $\texttt{x}_2$) no sólo se sincronizan mediante una cita, la ejecución de la función $f(IN\ x_1,\ IN/OUT\ x_2)$ se realiza localmente por el procesador que ejecuta el proceso $\texttt{P}_2$ pero cambia el estado del proceso remoto $\texttt{P}_1$.

(c) Después de ejecutar la función $f(IN\ x_1,\ IN/OUT\ x_2)$, el proceso $\texttt{P}_2$ envía los valores de los argumentos x y y de vuelta al proceso $\texttt{P}_1$.

(d) El proceso $\texttt{P}_2$ adquiere una dirección de memoria que referencia a la variable $\texttt{x}_2$ y que le envía el proceso $\texttt{P}_1$.En dicha dirección de memoria se asigna le valor final de la variable.

17. Responda Verdadero/Falso a las siguientes afirmaciones sobre las alternativas de la sentencia Select:

```
Proceso S1(){                    Proceso S2(){
  ENTRY E1();                      ENTRY E2();

   SELECT                           SELECT
                                     when C=>
    when C =>                          ACCEPT A;
    ACCEPT A;

  or DELAY 10.0;                   or DELAY 20.0;
      DELAY 10.0;
  END SELECT;                        END SELECT;
```

(a) La sentencia Select (S1) sólo terminaría si no se ha producido ninguna llamada a su punto de aceptación transcurridas 20.0 unidades de tiempo.

(b) Si la condición C de la sentencia Select (S1) se evalúa como falsa, entonces la ejecución de ambas sentencias Select es indistinguible.

(c) La sentencia Select (S1) siempre esperará 10.0 unidades de tiempo las llamadas a su punto de aceptación.

(d) Si no se producen llamadas a su punto de aceptación en ninguna de las sentencias Select anteriores en un plazo de 20.0 unidades de tiempo, el funcionamiento de ambas sentencias Select resulta ser el mismo.

18. Si existe un proceso $\texttt{P}_1$ que llama a un método remoto o.m(...) de un objeto o de la clase C, que se encuentra implementada en el ordenador donde se ejecuta el proceso $\texttt{P}_2$ (o *sirviente*). Para que la llamada del proceso $\texttt{P}_1$ al método remoto o.m(...) pueda se resuelta con el protocolo RMI, se ha de verificar una de las siguientes afirmaciones:

(a) Los procesos $\texttt{P}_1$ y $\texttt{P}_2$ han de ejecutarse en el mismo ordenador.

(b) El proceso $\texttt{P}_1$ sólo ha de conocer la interfaz remota que define los métodos públicos que implementa la clase C.

(c) El proceso $\texttt{P}_1$ ha de tener acceso siempre a un *stub* local del objeto remoto o, que entienda los mensajes que el proceso anterior envía al objeto o.

(d) Que el proceso P_1 tenga acceso a la interfaz remota aludida y también a 1 objeto *stub* que represente al objeto remoto de la clase que implementa el citado método.

19. Considérense las siguientes tareas en un lenguaje distribuido con invocaciones remotas:

```
1 Process P1
2 do{
3 P2.A;
4 P3.B;
5 P4.G;
6 P3.E;
7 accept F;
8 P3.C;
9 }while(true);
10End P1;
```

```
1 Process P2
2 do{
3  select
4   accept A do
5     P3.D;
6   or
7   accept H do
8    P3.E;
9 end select;
10 }while(true);
11End P2
```

```
1 Process P3
2 do{
3  accept D;
4  accept B;
5  accept E do
6  P1.F;
7 End P3;
8 accept C;
9 }while(true);
10End P3;
```

```
1 Process P4
2 do{
3  accept G;
4  T2.H;
5 }while(true);
6 End P4;
```

¿Puede producirse bloqueo global de los procesos anteriores? Si fuera así, indicar una secuencia de entrelazamiento que lleve a tal situación. Indicar el proceso y el número de línea de cada sentencia cuya ejecución provoque el bloqueo de cada proceso.

20. Completar el código de controlador de una máquina de bebidas, programando una selección no determinista, que se repite continuamente de acuerdo con las condiciones del problema:

(a) Para cada bebida, acepta peticiones de los clientes: `ACCEPT peticion(num_bebida : IN int)`.

(b) Puede llegar a tener en total N latas de bebidas disponibles de los diferentes tipos.

(c) Cuando se agota todo el suministro, la máquina entra en un periodo de mantenimiento durante 2 horas, tras el cual se reponen todas las latas y la máquina vuelve a estar en servicio.

(d) Cada 24 horas se reponen las latas que faltan si se ha consumido alguna; esta operación tarda en realizarse 1/2 hora.

```
Proceso Cliente (int i) {
  int pide;//5 tipos de bebidas
  pide= random()%5;
  do{
    Controlador.peticion(pide);
    //pasa a beberla antes de pedir otra
    delay random()*i%MAX;
  }while(true);
} Proceso Controlador{
   ENTRY peticion(num\_bebida : IN int);
   //Numero total de latas en la maquina
   int Latas=N;
  public void run(){
```

```
        do{
           SELECT
               /*
                completar
                */
            END SELECT;
        }while(true);
    }//fin Controlador
```

Fuentes Consultadas

[Andrews, 1991] Andrews, G. (1991). *Concurrent programming: principles and practice*. Benjamin Cummings, Redwood City, California.

[Barnes, 1994] Barnes, J. (1994). *Programming in Ada. Plus an Overview of Ada 9X*. Addison-Wesley, New-York.

[Dijkstra, 1975] Dijkstra, E. (1975). Guarded commands, nondeterminacy, and formal derivation of programs. *cacm*, 18(8):453:457.

[Hoare, 1985] Hoare, C. (1985). *Communicating sequential processes*. Prentice-Hall, London.

[Lea, 2001] Lea, D. (2001). *Programación concurrente en Java: principios y patrones de diseño*. Addison-Wesley/Pearson Education.

[Snir et al., 1999] Snir, M., Otto, S., Huss-Lederman, S., and D. Walker, J. D. (1999). *MPI: The Complete Reference*. The MIT Press, Cambridge, USA.

Capítulo 4

Introducción a los Sistemas de Tiempo Real

4.1 Introducción

En la actualidad el desarrollo de aplicaciones para STR tiene una gran importancia porque un amplio rango de sistemas poseen características de tiempo real [Burns, 2003]. Hay que tener en cuenta que en la actualidad el 99 % de la producción mundial de procesadores se utiliza para la construcción de sistemas empotrados, cuyo software de control suele basarse en un núcleo simplificado de operativo de tiempo real. A menudo, el tiempo real se confunde con *en línea*, con *interactivo* o con *rápido* [Stankovic, 1988]. Los responsables de la mercadotécnia de sistemas son particularmente propensos a este error. Que un sistema esté *en línea* significa que se encuentra siempre disponible, pero eso no garantiza la *responsividad*, es decir, una respuesta en un tiempo acotado. A menudo los sistemas en línea ni siquiera garantizan una respuesta. Por *interactivo* se entiende que el tiempo de respuesta del sistema es adecuado desde el punto de vista de un usuario humano. Si bien es cierto que habitualmente los sistemas en tiempo real trabajan a frecuencias altas para una persona (algunas del orden de cientos o miles de veces por segundo), no tiene por qué ser siempre así: un sistema complejo podría tener un plazo de segundos o incluso de días y aún operar en tiempo real. La condición de tiempo real es que la respuesta se obtenga en un plazo prefijado, independientemente de su duración.

En los STR continuos es preciso, además, que el tiempo necesario para procesar la información sea inferior al tiempo de su llegada. Los sistemas de procesamiento de audio, vídeo o telefonía sobre Internet son sistemas continuos de tiempo real. Si el tratamiento de una señal de sonido necesita 1,1 segundos por segundo de señal, el sistema no es de tiempo real; si en cambio bastan 0,9 segundos, entonces es posible hacer que sea un sistema de tiempo real, de tal forma que se podría escuchar la señal al tiempo que se va procesando.

La definición más comúnmente aceptada de lo que es un sistema de tiempo real (STR) es: "Un sistema en tiempo real es aquel en el que la respuesta correcta a un cálculo no sólo depende de su corrección lógica, sino también de cuándo dicha respuesta está disponible" [Burns, 2003]. Un buen ejemplo es el de un robot que necesita tomar una pieza de una banda sinfín. Si el

robot llega tarde, la pieza ya no estará donde debía recogerla. Por lo tanto el trabajo se llevó a cabo incorrectamente, aunque el robot haya llegado al lugar adecuado. Si el robot llega antes de que la pieza llegue, la pieza aún no estará ahí y el robot puede bloquear su paso.

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

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

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

4.1.1 Clasificación de los sistemas de tiempo real

Los STR se clasifican según su criticidad en permisivos o *suaves* (*"soft"*) y de misión crítica o no-permisivos (*hard*). Un sistema de *misión crítica* es aquel en que es inadmisible que los resultados lleguen tarde, mientras que en uno *permisivo* sólo se producirán pérdidas de rendimiento que, según su coste, pueden resultar aceptables en una versión entregable del sistema final. La pérdida de un tiempo límite supone un fallo total del sistema en el caso de los sistemas de *misión crítica*. Un sistema de misión crítica sería, por ejemplo, el encargado de controlar la maniobra de atraque del transbordador espacial Atlantis en la *International Space Station*, mientras que un reproductor de video que ocasionalmente pierde alguna trama puede ser un

ejemplo de sistema permisivo. El robot de la cinta de montaje de piezas de un ejemplo anterior exigiría un sistema de misión crítica si el llegar tarde a la pieza supusiese una paralización completa de la línea de montaje, mientras que podría ser permisivo si, como consecuencia de su retraso, disminuyese el ritmo de producción de la cadena de montaje.

Existe un tercer tipo de STR que se encuentra en medio de los dos anteriores (ver tabla 4.1), los denominados STR *estrictos* (*"firm"*). Con este tipo de sistemas, serían *tolerables* pérdidas infrecuentes del tiempo límite de las tareas de tiempo real, aunque dichas pérdidas pueden llegar a degradar la calidad de servicio del sistema. A diferencia de los STR permisivos, en los que se obtiene alguna ganancia con los resultados tardíos, la utilidad de los resultados que se producen después de cumplirse el tiempo límite es nula en los sistemas estrictos.

Los sistemas de tiempo real industriales suelen contener una mezcla de susbsistemas componentes de las tres clases anteriormente comentadas.

Denominación	Ejemplo	Complementos
Misión crítica	Control de aterrizaje	Tolerancia a fallos
Estrictos	Reservas de vuelos	Calidad de respuesta
Permisivos	Adquisición datos meteorológicos	Medidas de fiabilidad

Tabla 4.1: Clasificación de los sistemas de tiempo real atendiendo a su criticidad

4.1.2 Medidas de tiempo

La característica fundamental de un STR es su capacidad para ejecutar las instrucciones programadas en sus tareas dentro de intervalos de tiempo bien definidos. Por lo tanto, para poder desarrollar de forma apropiada cualquier STR hay que contar con mecanismos adecuados para la medida del tiempo, así como para controlar la duración de las instrucciones referidas anteriormente [Cheng, 2002].

En la actualidad resulta impensable desarrollar software con características de tiempo real sin contar con el apoyo de un lenguaje de programación de alto nivel y un sistema operativo apropiado , que nos proporcionen: (a) relojes de *tiempo real*, que nos ayuden a medir el tiempo con precisión; (b) mecanismos para activar tareas en instantes previamente determinados; (c) *tiempos límite de espera* ("timeouts"[1]); (d) planificadores de tareas de tiempo real adaptables por el programador a las necesidades de su aplicación.

El tiempo es una magnitud física fundamental, cuya unidad en el Sistema Internacional (SI) es el segundo. Para poder desarrollar software necesitaremos dos tipos de medidas del tiempo:

- Tiempo absoluto
- Intervalos o tiempo relativo

[1] un periodo de tiempo después del cual surge una condición de error si no se ha producido algún evento, se ha recibido una entrada, etc. Un ejemplo frecuente es el envío de un mensaje. Si el receptor no reconoce el mensaje dentro de un periodo de tiempo prestablecido, se infiere que ha ocurrido un error de transmisión y se levanta una excepción en el programa.

El tiempo absoluto necesita un sistema de referencia con un origen que se denomina *época*. Podemos utilizar varios sistemas de referencia: (a) *locales*: suelen coincidir con el tiempo transcurrido desde el arranque de nuestro sistema; (b) *astronómicos*: por ejemplo, el denominado *Tiempo Universal* (UT0), que es el término moderno usado para la medida internacional de tiempo utilizando un sistema basado en telescopio, y que fue adoptado en 1928 para reemplazar al sistema GMT ("Greenwich Mean Time") por la Unión Astronómica Internacional; (c) *atómicos*: el tiempo se mide en Ciencia con una cuenta continua de segundos basada en relojes atómicos ubicados alrededor del mundo, este sistema se conoce con el nombre de Tiempo Atómico Internacional (IAT); la duración de 1 segundo es constante, puesto que es definida a partir del inmutable periodo de transición del átomo de Cesio; (d) El *Tiempo Coordinado Universal* (UTC): actualmente la base para la medida del tiempo en la vida civil, desde el 1 de Enero de 1972, en que fue definido para seguir al IAT con una desviación dada por un número entero de segundos, cambiando sólo cuando se le añade 1 segundo para sincronizarse con la rotatión de la Tierra; (e) *satelital*: el Sistema de Posicionamiento Global (GPS) también transmite una *señal de tiempo* para todo el planeta, además de proporcionar instrucciones precisas para convertir el tiempo GPS a UTC.

Relojes de tiempo real

Un *reloj* en el contexto de esta asignatura es un módulo compuesto por elementos *hardware* y *software* que nos proporciona el tiempo real cuando se lee su valor, que mantiene actualizado durante la ejecución de nuestro sistema. Un *reloj* está compuesto por: (a) un circuito *oscilador* que genera impulsos eléctricos, (b) un *contador* que acumula los impulsos guardando su valor actualizado en una palabra específica en memoria, (c) un *software* que convierte el valor del contador a unidades de tiempo definidas en el SI. Las características más importantes de un reloj son:

- *Precisión*, es decir, cada cuánto llega un nuevo impulso que cambia la cuenta que lleva acumulada, también se conoce como *granularidad* o valor del menor *grano* de tiempo que es capaz de discriminar. La precisión dependerá de la frecuencia de su oscilador y de la forma de detectar y contar los impulsos.
- *Intervalo*: el mayor rango de valores de tiempo que es capaz de medir antes del *desbordamiento*. Esta característica depende directamente de la precisión del reloj y de la capacidad del contador. A mayor precisión, o menor grano de tiempo del reloj, se obtendrá un menor tamaño[2] del intervalo de valores para una capacidad constante del contador.

Dado que la capacidad del contador es limitada, al llegar a un valor determinado de la cuenta de impulsos acumulada se producirá un reinicio de dicha cuenta. A esto se le conoce como *desbordamiento* del contador del reloj de tiempo real. El principal efecto que tiene sobre la medida del tiempo en una aplicación de tiempo real consiste en el establecimiento de dos escalas temporales:

- Tiempo monótono: la *vida* de la aplicación coincide con el valor máximo acumulado en el contador, ya que la aplicación de tiempo real posee un tiempo de ejecución menor que el tiempo de desbordamiento del reloj.

[2] 1ns=$10^{-9}s$,1$\mu\, s = 10^{-6}s$, 1ms=$10^{-3}s$

Precisión	Intervalo
100 ns.	Hasta 429,5 s.
1 μs.	Hasta 71,58 m.
100 μs.	Hasta 119,3 h.
1 ms.	Hasta 49,71 días
1 s.	Hasta 136,18 años

Tabla 4.2: Tamaño del intervalo vs.precisión para un contador de 32 bits.

- Tiempo no monótono: en este caso, como se puede ver en la figura 4.1, la aplicación dura más que el tiempo de desbordamiento, pudiéndose obtener el tiempo total como la suma de varias cuentas del contador.

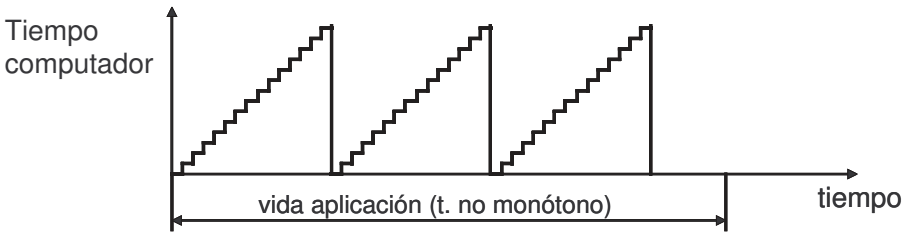


Figura 4.1: Representación del tiempo no-monótono en un reloj de tiempo real.

Los problemas que se derivan de la utilización de una escala de tiempo no–monótono se concretan en que no se dispone de una medida de tiempo *absoluta*, es decir, con un único origen temporal, y no se puede mantener la fecha, la hora, etc., ya que se reinicia el valor del contador varias veces. Tampoco se pueden utilizar intervalos temporales que se extiendan más allá del valor de desbordamiento del contador del reloj.

Temporizadores y retardos

Un temporizador (o *"timer"*) es un tipo de reloj especializado. Los sistemas operativos a menudo utilizan un único temporizador–hardware para implementar un conjunto extensible de temporizadores-*software*. En este escenario, la rutina de servicio de interrupción del reloj hardware manejaría el funcionamiento y la gestión de tantos temporizadores–software como hicieran falta, y su plazo se ajustaría para expirar cuando el siguiente temporizador–software haya de hacerlo.

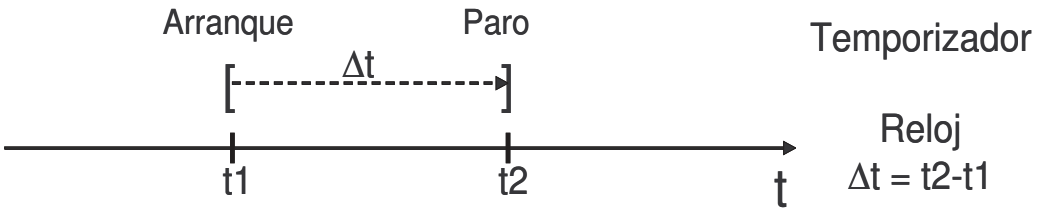


Figura 4.2: Temporizador–software programable

En cada expiración del plazo del temporizador–hardware se comprueba si ha vencido el plazo del siguiente temporizador–software, así como se iniciarían sus acciones pendientes.

Los temporizadores pueden ser utilizados para controlar la secuencia de actuación de un proceso o las consecuencias de la aparición de un evento del entorno de una aplicación de tiempo real. Su utilidad fundamental en las aplicaciones es la de medir intervalos temporales. Para programar un temporizador–software hay que indicar sus tiempos de *arranque* y *parada*, ver figura 4.2. Los temporizadores pueden ser de un solo disparo o periódicos. Los temporizadores de un solo disparo interrumpen una única vez, y después se paran definitivamente. Los temporizadores periódicos interrumpen cada vez que se alcanza un valor temporal específico. Esta interrupción es recibida a intervalos regulares desde el temporizador–hardware. El manejo de los temporizadores parece algo sencillo, sin embargo, hay que tener cuidado con aspectos tales como la *deriva* o las interrupciones retrasadas, que han de ser minimizados, cuando se implementan temporizadores–software, o de otra forma no conseguiríamos precisión en la activación de las acciones de la aplicación que dependan de temporizadores.

Los *retardos* permiten controlar el tiempo de activación de las tareas de tiempo real de una aplicación. Pueden programarse utilizado temporizadores, a nivel de sistema operativo; o bien, mediante instrucciones de los lenguajes de programación. En este último caso se conseguirá una máxima *transportabilidad*[3] del tiempo de retraso especificado. La forma de programarlos sería similar a: delay < duración >; , o bien, con más precisión: *nanosleep*< duración >; . Su efecto consiste en suspender la ejecución de la tarea durante, al menos, la duración especificada desde el momento en que se produjo la llamada a la instrucción anterior. La ejecución puede verse retrasada durante un tiempo mayor que el especificado en la llamada debido a una menor precisión del reloj, que la necesaria para servir la llamada con exactitud; o bien debido a que se activa una tarea más prioritaria cuando termina el tiempo del retardo y la tarea retrasada se ve pospuesta en el acceso al procesador.

La programación con retardos de tareas que se activan periódicamente podría causar la *deriva acumulativa* de la activación de las tareas en cada nuevo ciclo. Esto es debido a que los retrasos que se producirían en cada periodo, es decir, la *deriva local*, se van acumulando.

```
tarea periodica::
  periodo= 100; //milisegundos
  do {
    delay periodo; // en cada ciclo produce un retardo = 'periodo'
    // accion a realizar
  while (true);
  }
```

Figura 4.3: Tarea periódica afectada de deriva local

Se puede conseguir la eliminación de la deriva acumulativa de la programación de tareas periódicas haciendo que, en cada ciclo, la tarea se retrase hasta el siguiente instante de activación. De esta forma, cada activación de la tarea puede significar un tiempo efectivo de retraso diferente, ya que se descontarán los tiempos durante los cuales la tarea está desplazada del acceso al procesador por causas ajenas al propio retardo.

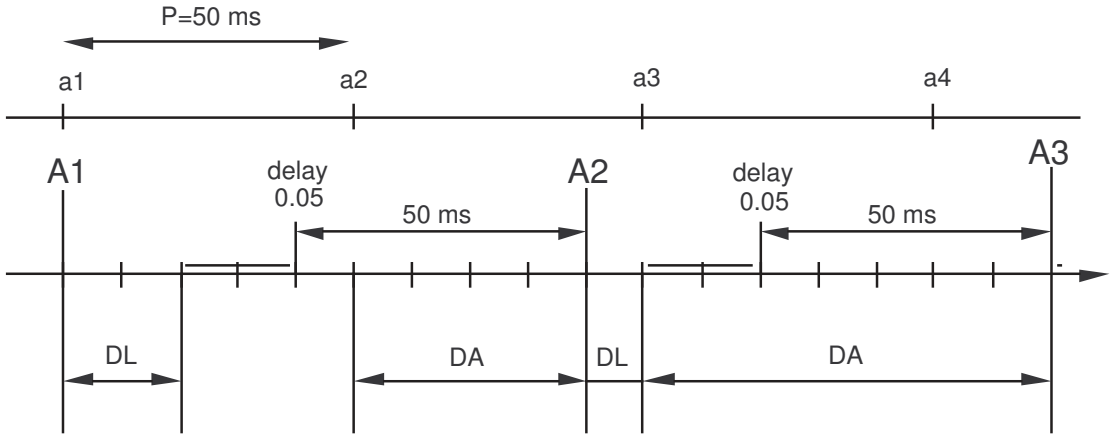
Tal como se puede ver en la siguiente figura, si en cada ciclo la tarea periódica experimenta un retraso constante ("primera aproximación"), la deriva acumulativa (DA) se irá incremen-

[3]es decir, para el código programado, se obtendría el mismo retraso, independientemente del sistema o de la plataforma de ejecución de la aplicación

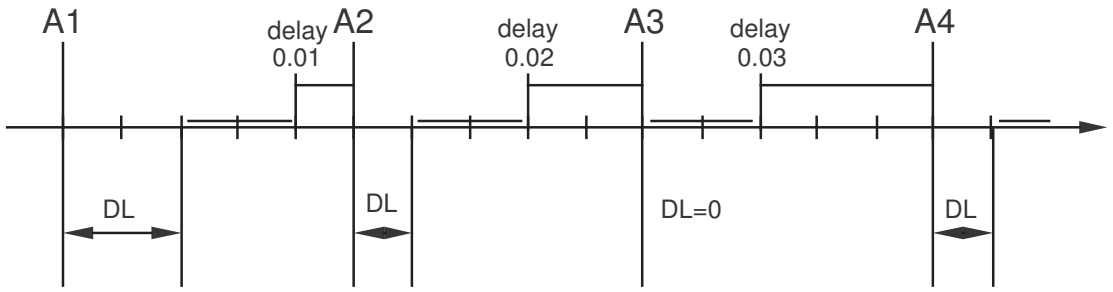
```
tarea periodica::
  periodo= 100; //milisegundos
  siguiente_instante= 0; //milisegundos
  ...
  siguiente_instante= clock();//funcion del sistema
  do {
    delay (siguiente_instante - clock());
    // accion a realizar
    siguiente_instante += periodo;
  while (true);
  }
```

Figura 4.4: Tarea periódica con deriva local eliminada

tando. Si, por el contrario, los retrasos son variables, para ajustarse al siguiente instante de expiración del retardo, entonces se consigue eliminar la deriva acumulativa ("aproximación buena").



Primera aproximación



Aproximación buena

Figura 4.5: Derivas en las tareas con retardos.

Tiempos límite de espera

A veces resulta necesario limitar el tiempo durante el cual una tarea espera que ocurra algún evento. Existen múltiples estados de las tareas durante los cuales éstas quedan suspendidas, por ejemplo, cuando solicita la realización de una operación de entrada/salida u otro servicio del sistema operativo, que se realiza de una forma síncrona con la ejecución de dicha tarea. En estos casos, si la tarea que coopera con la primera para realizar el servicio, o bien el dispositivo, fallasen, entonces la tarea suspendida no volverá a estar activa nunca más. La solución para evitarlo es programar la operación sujeta a un *tiempo límite de espera* (o "timeout"), que indicará el tiempo máximo durante el cual permanecerá suspendida. Si transcurrido dicho tiempo no se realiza el servicio, entonces la tarea suspendida vuelve, levantándose una excepción que puede ser utilizada para presentar un mensaje de aviso o devolver un código de error.

4.1.3 Modelo de tareas

Para poder analizar de una forma comprensible el comportamiento de una tarea durante su ejecución, en el peor caso posible de planificación[4], dentro un programa o aplicación de tiempo real arbitrariamente compleja, es necesario imponer algunas restricciones a su estructura e interacciones con el resto de tareas del programa. Las tareas de tiempo real que son conformes con las restricciones impuestas se dice que cumplen con el *modelo de tareas simple* [Buttazzo, 2005].

Aunque es un modelo básico, el modelo de tareas siemple tiene la capacidad descriptiva suficiente para que se puedan modelar esquemas de planificación estándar, tales como el basado en asignación estática de prioridad, que es el más utilizado en el desarrollo y análisis de STR hasta la fecha.

Características del modelo simple

Consideramos un programa de tiempo real como un conjunto fijo[5] de tareas, que se ejecutan compartiendo el tiempo de un solo procesador, es decir, (*concurrentemente*).

Las tareas son periódicas, con periodos conocidos e independientes entre sí. No existen semáforos, objetos compartidos, etc. que pudieran bloquear a una tarea más prioritaria debido a que el recurso que pretende bloquear ha sido ya adquirido por otra, posiblemente de menor prioridad que ella.

Todas las tareas poseen un tiempo límite (*deadline*), que se considera igual a su periodo. Una tarea, por tanto, está obligada a terminar completamente su ejecución antes de la siguiente activación, lo que ocurrirá cuando transcurra un tiempo igual a su periodo desde el instante de inicio de su activación actual.

Las sobrecargas o retrasos que pueda experimentar el sistema, por ejemplo, tiempos de cambio de contexto, etc. son ignorados. Suponemos que nada impide a una tarea en estado

[4]esto es, cuando sufre una mayor interferencia por parte de procesos más prioritarios
[5]durante la ejecución del programa no se crean ni se destruyem tareas

ejecutable obtener el procesador si en un determinado momento de la aplicación pasa a ser la tarea más prioritaria.

Los eventos no son almacenados, es decir, se pierden si no se atienden y el tiempo máximo de cómputo que una tarea necesita para ser procesada es fijo y conocido a-priori para cada proceso. Se denota con el símbolo `C`, el cual representa unidades de tiempo, y se le denomina "tiempo de ejecución de peor caso" (o WCET, según su acrónimo en inglés).

En este modelo no se contempla a las tareas esporádicas, que ocurren ocasionalmente durante la ejecución de una aplicación de tiempo real y que suelen tener una gran urgencia cuando se activan.

Atributos temporales asociados a una tarea

Una tarea de tiempo real τ_i puede ser caracterizada mediante un conjunto específico de atributos temporales.

En la figura 4.6 se puede observar una representación gráfica de algunos de los atributos anteriores, que se consideran como elementos característicos dentro de un esquema de planificación de tareas en un STR.

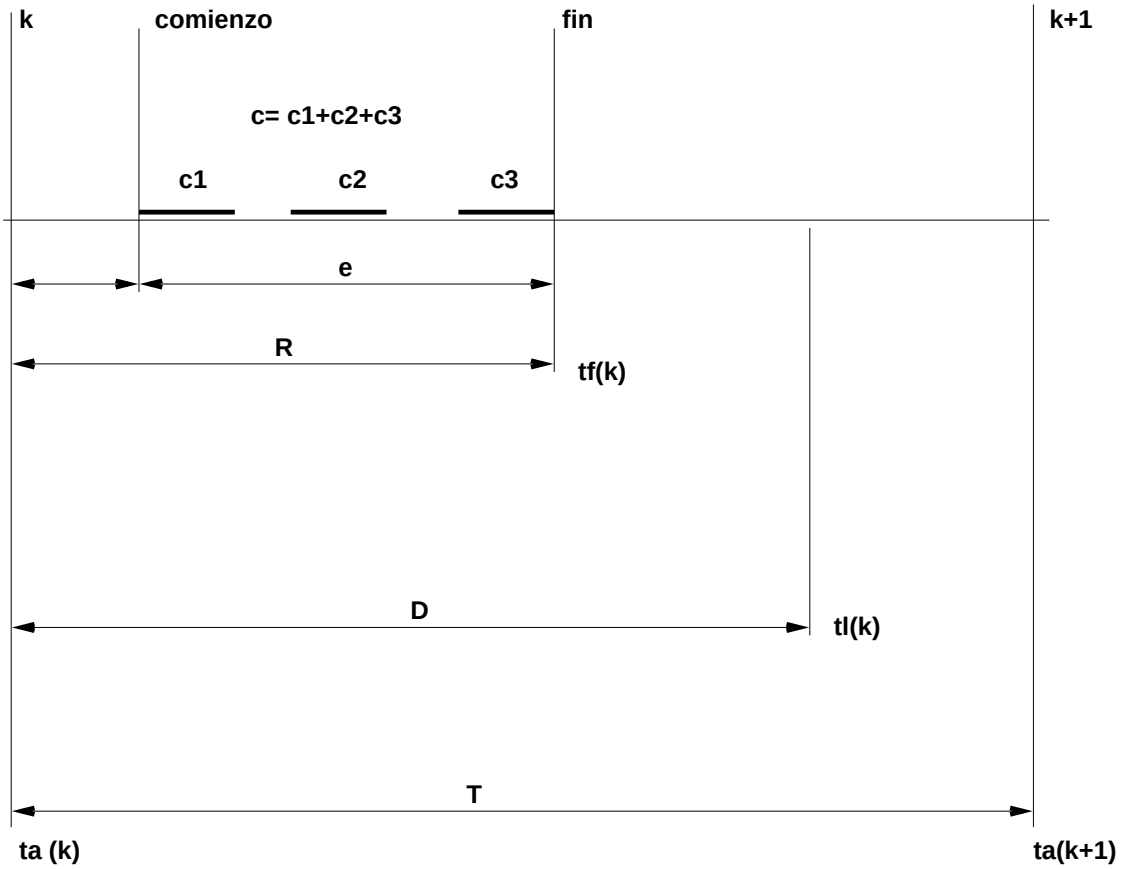


Figura 4.6: Representación de los atributos temporales de las tareas de tiempo real.

Notación	Atributo temporal	Descripción
P	Prioridad	Prioridad asignada al proceso (si fuera aplicable)
τ	Tarea	Nombre de la tarea
t_a	Instante o tiempo de activación de la tarea (arrival time, request time, release time)	Instante en el que la tarea está lista para su ejecución.
t_s	Instante o tiempo de comienzo (start time)	Instante de tiempo en el que la tarea comienza realmente su ejecución.
t_f	Instante o tiempo de finalización de la tarea (finishing time)	Instante en el que la tarea finaliza su ejecución.
t_l, d	Instante o tiempo límite (absolute deadline)	Instante de tiempo límite para la ejecución de la tarea. Es un valor fijo dado por $t_l(k) = t_a + D$
T	Periodo de ejecución	Intervalo de tiempo entre dos activaciones sucesivas de una tarea periódica. Es un valor fijo, dado por $T = t_a(k+1) - t_a(k)$
J	Latencia	Intervalo de tiempo desde que se activa la tarea hasta que se ejecuta. Viene dado por: $J(k) = t_s(k) - t_a(k)$ y, dependiendo de la sobrecarga del sistema, varía entre un valor mínimo de ($J_{mín}$) y máximo de (J_{max})
c	Tiempo de cómputo	Tiempo de ejecución del proceso
C	Tiempo de cómputo máximo	Tiempo de ejecución del proceso en el peor caso posible.
e	Tiempo de ejecución transcurrido	Tiempo transcurrido desde el instante de comienzo hasta la finalización del proceso. Viene dado por: $e(k) = t_f(k) - t_s(k)$
R	Tiempo de respuesta	Tiempo que ha necesitado el proceso para completarse totalmente. Es variable en cada activación de la tarea y viene dado por: $R(k) = J(k) + e(k)$.
D	Plazo de respuesta máximo (relative deadline)	Define el máximo intervalo de tiempo o máximo tiempo de respuesta hasta completar la ejecución de la tarea.
Φ	Desplazamiento o fase	Tiempo necesario para activar por primera vez una tarea periódica.
RJ	Fluctuación relativa o *jitter* (relative release jitter)	Máxima desviación en el tiempo de comienzo entre dos activaciones sucesivas de una tarea. Viene definido por: $RJ = \max((t_s(k+1) - t_a(k+1)) - (t_s(k) - t_a(k)))$
AJ	Fluctuación absoluta	Máxima desviación en el tiempo de comienzo de todas las activaciones de una tarea. Viene definido por: $AJ = J_max - J_min$
L	Retraso(lateness). Existe también el tiempo de *exceso*, dado por: $E(k) = \max(0, L(k))$	Retraso de la finalización del proceso respecto del tiempo límite. Viene definido por: $L(k) = t_f(k) - t_l(k)$. Si la tarea se completa antes del tiempo límite su valor es 0.
H	Holgura (laxity, slack time)	Tiempo máximo que una tarea puede permanecer activa dentro del plazo de respuesta máximo. Se define como: $H(k) = t_l - t_a(k) - c(k) = D - c(k)$

4.2 Planificación de tareas periódicas con asignación de prioridades

De forma general se puede definir la *planificación* de actividades o tareas como un área del conocimiento humano que estudia un conjunto de algoritmos y técnicas de *programación entera* cuyo objetivo es conseguir una asignación de recursos y tiempo a actividades, de forma que se cumplan determinados requisitos de eficiencia. La estrategia básica para resolver un problema de planificación de recursos suele ser utilizar una heurística que intenta maximizar una función objetivo.

En el caso de los sistemas de tiempo real, el principal recurso a asignar es normalmente el tiempo del procesador. Se suele requerir, además, que sea posible determinar *a-priori* si las tareas de un programa terminan, en todas sus activaciones, antes de que se alcancen sus tiempos límite, aun en el peor caso de planificación posible de dichos procesos.

La determinación de la planificabilidad de un conjunto de procesos se lleva a cabo utilizando un *esquema de planificación de procesos* que ha de contener los siguientes elementos:

1. Un algoritmo para ordenar el acceso de las tareas a los recursos del sistema.

2. Una forma de predecir el comportamiento del sistema en el peor de los casos.

Existen diferentes tipos de esquemas de planificación de tareas de tiempo real. Se habla, por ejemplo, de esquema de planificación *estático* cuando el orden de planificación de las tareas es fijo y puede ser, por tanto, determinado a-priori. Por contra, hablaremos de un esquema de planificación *dinámico* cuando la prioridad de los tareas varíe a lo largo de la ejecución. Con un esquema estático simplificamos el problema de la planificación de tareas de tiempo real, sin perder generalidad. Este modelo de tareas es seguido por la mayoría de la aplicaciones de tiempo real en sistemas muy críticos, como los que se utilizan en el control de vuelo de un avión comercial o en radio–medicina.

En modelos más elaborados de tareas habría que tener en cuenta a las tareas no–periódicas, ya que cuando ocurren suelen ser tareas muy urgentes y críticas para la seguridad del STR. De acuerdo con el modelo simple de tareas, no se van a tratar las tareas de tipo *esporádico* o aperiódico. Es decir aquellas tareas que cuando se activan tienen un tiempo límite mucho menor que su periodo o que carecen en absoluto de un patrón periódico de activación, respectivamente.

Se supone un esquema de planificación expulsivo (*preemptive*), es decir, se produce siempre un *desplazamiento* del procesador de las tareas menos prioritarias por parte de los más prioritarias. Para determinar si las tareas $\tau_1\ \tau_2\ \ldots\ \tau_n$, con periodos $T_1\ T_2\ \ldots\ T_n$, pueden ser totalmente ejecutadas dentro de sus plazos temporales, la línea temporal del gráfico de planificación habría de cubrir un periodo de tiempo de *al menos* igual al **m.c.m.**$(T_1\ T_2\ \ldots\ T_n)$. Aunque sólo será necesario analizar una línea temporal de longitud correspondiente al mayor T_i si se supone que todas las tareas se inician a la vez. Este instante representa un momento de la *máxima carga* posible del procesador; por tanto, se le denomina (*instante crítico*).

Conforme progresa la aplicación, las tareas se ejecutan en el orden dado por su prioridad. Cuando se utiliza este esquema, los parámetros que influyen en la planificación de las tareas se

tienen que fijar *estáticamente*. Por tanto, el máximo tiempo de ejecución de las tareas es fijo, así como lo son también sus prioridades asignadas.

La *prioridad* es un número positivo. Normalmente se toma la convención de asignar números enteros menores a los procesos más prioritarios. La prioridad de una tarea se determina a partir de su periodo, tiempo límite, etc., o cualquier otro atributo que se mantenga durante toda su ejecución.

4.2.1 Algoritmo de cadencia monótona

Aquí nos centraremos en un esquema de planificación estático para tareas periódicas, en el que la prioridad de una tarea de tiempo real sólo dependerá de su periodo. Es decir, se asignarán inicialmente las prioridades a las tareas de la aplicación en el orden dado por su menor frecuencia de activación. Las tareas con periodos de activación más cortos van a ser las más prioritarias, independientemente de su criticidad respecto de la aplicación a la que pertenecen. Matemáticamente, podemos decir que dichas prioridades se asignan mediante una función monótona de la cadencia temporal de los procesos periódicos: $T_i < T_j \Rightarrow P_i > P_j$, de ahí el nombre del algoritmo.

El algoritmo de *cadencia monótona* es óptimo entre los algoritmos de asignación estática de prioridades, esto es, un conjunto de procesos planificable con cualquier esquema de asignación fija de prioridades resultaría también planificable si en lugar de este se utilizase el algoritmo de cadencia monótona.

El algoritmo, propuesto en un trabajo histórico por Liu y Layland en 1973, es la base para desarrollar una teoría matemática de planificación de tareas de tiempo real. Dicha teoría dista mucho de ser algo exclusivamente teórico, ya que tiene en cuenta aspectos prácticos de la planificación de tiempo real impuestos por la cooperación con la industria norteamericana. De hecho, gran parte de los resultados se obtuvieron como consecuencia de la colaboración entre 3 instituciones: Software Engineering Institute (SEI), IBM y Carnegie Mellon University.

4.2.2 Tests de planificabilidad

Para poder determinar, antes de su ejecución, si un conjunto de tareas periódicas es planificable utilizando el algoritmo de cadencia monótona es necesario contar con criterios que permitan predecirlo. Dichos criterios se establecen en función de la utilización del procesador (U) por parte del conjunto de tareas, o bien calculando el tiempo de respuesta de cada tarea (R_i), los cuales pueden ser calculados estáticamente.

La obtención de tests de planificabilidad para un conjunto de tareas con prioridades asignadas estáticamente no ha sido fácil. De hecho, hasta muy recientemente, no se han descubierto condiciones suficientes y necesarias para determinar la planificabilidad de las tareas de un programa de tiempo real arbitrario.

Se va a comenzar estudiando algunas *condiciones necesarias*, que intuitivamente deberían cumplirse, para poder afirmar que un conjunto de tareas es planificable. Dichas condiciones

pueden utilizar datos estáticos de las tareas: peor tiempo de ejecución, periodo, etc. para poder ser definidas.

Condiciones necesarias de planificabilidad

1. El tiempo de ejecución en el peor de los casos de cualquier tarea ha de ser menor que su periodo.

   $$\forall\ i\ C_i\ <\ T_i$$

   Esta condición no es *suficiente*, ya que alguna de las otras tareas más prioritarias puede no haber terminado cuando llegue su tiempo límite. Por ejemplo, considérense:

-	Prio	T_i	C_i
τ_1	2	10	8
τ_2	1	5	3

   En el conjunto de tareas anterior, τ_2 perderá su primer tiempo límite en t=5. Sin embargo, se cumple el criterio 1.

2. La utilización del procesador por unidad de tiempo de todas las tareas con una prioridad superior a la del *nivel i* no puede superar la unidad. $\sum_{j=1}^{i}\ \frac{C_j}{T_j}\ \leq\ 1$. Esta condición no es suficiente, ya las tareas más prioritarios pueden interferir varias veces y hacer perder el límite de tiempo al proceso de nivel de prioridad i. Compruébese esto para el siguiente conjunto de tareas:

-	Prio	T_i	C_i
τ_1	2	6	3
τ_2	2	9	2
τ_3	1	11	2

   A τ_3 le faltaría por ejecutar una unidad de tiempo cuando ocurre el siguiente evento de activación y, como $\mathrm{D_i} = \mathrm{T_i}$, pierde su tiempo límite.

3. El procesamiento de todas las invocaciones en los niveles de prioridad mayores que i: 1, 2, ... , $(i-1)$, debe completarse como mucho en el tiempo $T_i\ -\ C_i$. Con referencia al ejemplo anterior, esto supondría que todas las activaciones de las tareas τ_1 y τ_2 que interfieren a τ_3 deberían completarse antes de 9 unidades de tiempo: $\forall\ i\ \ \sum_{j=1}^{i-1}(\frac{T_i}{T_j} \times C_j) \leq\ T_i\ -\ C_i$. Esta condición no es suficiente, ya que si $T_j\ >\ T_i$, entonces degenera en la primera condición (téngase en cuenta que las divisiones son enteras).

4. Para evitar que se puedan anular términos de la sumatoria que aparecen en la condición anterior, se amplia la *ventana temporal* de un periodo $\mathrm{T_i}$ a un marco temporal M_i = **m.c.m.**$\{T_1\ T_2\ ...\ T_i\}$, de esta forma se obtiene ahora el número exacto de activaciones de cada tarea T_j, más prioritaria que T_i, en el periodo de tiempo M_i, como el resultado de la fracción entera: $\frac{M_i}{T_j}$. La nueva condición, que adquiere el nombre de *razón de carga*, se expresa ahora como: $\forall\ i\ \ \sum_{j=1}^{i}(\frac{M_i/T_j}{M_i} \times C_j) \leq\ 1$. Sin embargo, aunque discrimina más que la condición 3, tampoco es una condición suficiente. Ya que si el tiempo de computación de una tarea τ_i excede al periodo de activación de otra τ_j, dado $i\ <\ j$,

entonces no sería factible la planificación conjunta ambas. Compruébese para el siguiente conjunto:

-	Prio	T_i	C_i
τ_1	2	12	5
τ_2	1	4	2

Condiciones suficientes de planificabilidad

Teorema 1 (Liu y Layland):

En un sistema de N tareas periódicas independientes con prioridades asignadas en orden de frecuencia[6], se cumplen todos los plazos de respuesta, para cualquier desfase inicial de las tareas, si:

$$\sum_{i=1}^{N}(\frac{C_i}{T_i}) < N\,(2^{\frac{1}{N}} - 1)$$

Si una tarea pasa el test anterior, entonces completará sus C_i unidades de cómputo antes de que expire el tiempo límite ($D_i = T_i$) en cada ciclo de activación de la tarea. Aun si falla el test anterior, podrían satisfacerse los tiempos límite de las tareas del conjunto anterior, porque se trata de una *condición suficiente*, pero no necesaria, de planificabilidad. Para comprobarlo se puede realizar un diagrama de ejecución en función del tiempo de todas las tareas, también denominado diagrama de Gantt. Si ninguna de las tareas perdiese su tiempo límite dentro de una ventana temporal dada por el $m.c.m\ de: T_1, T_2, ...T_i$ entonces podríamos afirmar la planificabilidad del conjunto, incluso si no se cumpliera el teorema anterior. El problema con esto es que la verificación de la planificabilidad de un conjunto grande de tareas en aplicaciones complejas es infactible salvo si cuenta con herramientas gráficas apropiadas, que suelen ser muy costosas.

A continuación, se puede ver una tabla con los límites de utilización del procesador para diferentes números de tareas:

N	límite utilización
1	100
2	82,85%
3	78,0%
4	75,7%
5	74,3%
10	71,8%
$\rightarrow \infty$	69,3%

[6]esto es, con el *algoritmo de cadencia monótona*

Por tanto, siempre que el límite de utilización del procesador (U), por parte de un conjunto de tareas, no supere el 69,3%, dicho conjunto será planificable utilizando un esquema de planificación expulsivo basado en la asignación estática de prioridades, dado por el algoritmo de la cadencia monótona. El test de Liu-Layland no es *exacto*, ya que dicho test, basado en la *utilización del procesador* (U), no es capaz de determinar la planificabilidad en algunos casos de un conjunto de tareas. El conjunto de tareas siguiente no pasa el test de Liu-Layland, ya que la utilización del procesador sería: $\Sigma_{i=1}^{3}(\frac{C_i}{T_i}) = .92857$ y, sin embargo, el test exige una utilización máxima del procesador $3 * (2^{\frac{1}{3}} - 1) = .77976$ como condición suficiente para asegurar la planificabilidad. Sin embargo, en la realidad, todos las tareas consiguen alcanzar sus tiempos límite:

-	Prio	T_i	C_i
τ_1	1	7	3
τ_2	1	12	3
τ_3	2	20	5

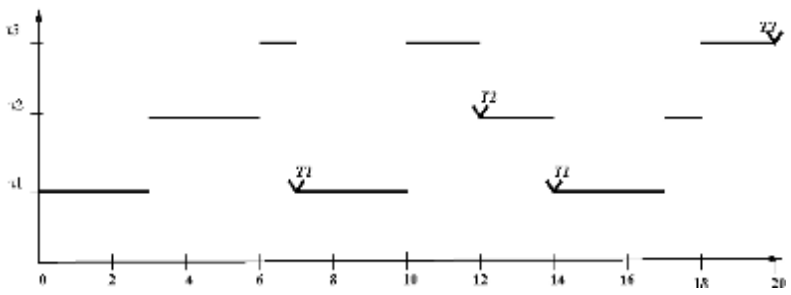


Figura 4.7: Conjunto planificable que no cumple el test de Liu

Además, el test de Liu no es aplicable, sin modificación, si se considera un modelo de tareas más general en el cual pueda existir aperiodicidad, bloqueos de tareas prioritarias, etc.

Los tests basados en la *utilización del procesador* no dan información acerca de los tiempos de respuesta de los procesos. De ahí que se propongan nuevos tests exactos, es decir, que proporcionan condiciones necesarias y suficientes de planificabilidad, basados en el cálculo de dichos tiempos de respuesta de las tareas de tiempoo real. El inconveniente consiste en su aplicación práctica porque el cálculo para un conjunto arbitario de tareas, que se pueden interrumpir unas a otras varias veces durante su ejecución, no tiene, en general, una solución matemática sencilla.

Teorema 2 (Liu y Layland):

> En un sistema de N tareas periódicas independientes con prioridades estáticas, se cumplen todos los tiempos límite, para cualquier desfase inicial de las tareas, si cuando se activan todas ellas simultáneamente, cada tarea acaba antes de que expire su tiempo límite en su primera activación.

4.2.3 Test de planificabilidad basado en la utilización para EDF

En el trabajo de Liu y Layland se propuso también un test para un esquema de asignación de prioridades basado en la *proximidad del tiempo límite* (EDF) de las tareas:

$$\Sigma_{i=1}^{N}(\frac{C_i}{T_i}) \leq 1$$

Si el límite de utilización derivado de la ejecución del conjunto de tareas es menor que la capacidad de cómputo total del procesador, entonces podemos afirmar que, para el modelo de procesos simple, se cumplirán todos los tiempos límite de las tareas.

El esquema de planificación EDF se considera dinámico, porque la prioridad de las tareas cambia conforme se acerca su tiempo límite durante la ejecución. EDF es superior a uno basado en la asignación estática de prioridades, ya que siempre que se pueda planificar un conjunto de tareas con un esquema estático, también se podrá con el EDF, pero no se cumple la inversa. No obstante, se prefieren los esquemas de planificación estáticos, ya que:

- Un esquema estático es más sencillo de implementar. Un esquema dinámico, como el EDF, induce mayor sobrecarga al sistema.
- Resulta más sencillo el incorporar tareas sin tiempos límite definidos en un esquema estático.
- El atributo de tiempo límite suele no ser el único parámetro de planificación a considerar en aplicaciones realistas. Resulta más sencillo incorporar otros factores que influyen en la planificación a la noción de prioridad cuando ésta no está asociada a un tiempo límite.
- Durante situaciones de sobrecarga transitoria un esquema de planificación estático resulta ser más predecible[7]. Además se suelen conseguir mayores límites de utilización del procesador con un esquema estático.

4.3 Modelos generales y específicos de tareas

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

Los tareas periódicas se ejecutan tras producirse eventos de activación locales. Las *tareas esporádicas* se ejecutan tras producirse un evento que se origina en un procesador remoto. El periodo T_i de una tarea τ_i esporádica se define como el mínimo[8] intervalo temporal medido entre dos eventos de activación sucesivos. El periodo de una aperiódica es el intervalo temporal promedio entre 2 eventos cualesquiera de activación sucesivos. Las tareas esporádicas, cuando

[7] las tareas con menor prioridad son las que pierden sus tiempos límite; sin embargo, esto no es necesariamente así si se utiliza un esquema EDF

[8] una tarea esporádica con $T_s = 20$ms. garantiza que no será activada más de 1 vez cada 20 ms. El intervalo entre 2 activaciones podría ser mayor algunas veces, pero ha de contemplarse el *peor caso* de planificación.

se presentan, suelen ser urgentes y tienen límites de tiempo estrictos mucho menores que su periodo de activación ($D_i << T_i$). Las tareas *aperiódicas* no tienen definido ningún tiempo límite estricto, como ocurre con las *esporádicas*. Suelen tener límites de tiempo permisivos, es decir, se admite que pierdan alguno de estos; a diferencia de las tareas esporádicas, en las cuales la pérdida de algún tiempo límite no sería admisible.

En aplicaciones de tiempo real que posean tareas esporádicas no se puede utilizar directamente el test de Liu para decidir la planificabilidad de un grupo de tareas. Ya que la utilización del tiempo del procesador, como criterio de planificabilidad de las tareas, suele propocionar un resultado demasiado "pesimista" cuando se le aplica a las esporádicas. Para este tipo de tareas se define una cadencia máxima y promedio de llegada de sus eventos de activación. El considerar el valor máximo suele conducir a la imposición de utilizaciones muy bajas del tiempo del procesador para garantizar la condición de planificabilidad del conjunto de tareas esporádicas.

4.3.1 Plazos de respuesta menores que el periodo

Para valores del tiempo límite que coincida con el periodo de las tareas, $D_i = T_i$, la asignación de prioridades según el *algoritmo de la cadencia monótona* resulta ser óptima para cualquier esquema de planificación estático. De forma análoga, el criterio de ordenación de las prioridades de las tareas atendiendo al menor valor de sus tiempos límite (*Deadline Monotonic Priority Ordering*) es *óptimo*[9] para conjuntos de tareas en los que se cumpla $D_i < T_i$. Un ejemplo en el que se cumple la afirmación anterior puede verse en la siguiente tabla:

τ	T	D	C	P	R
a	20	5	3	1	3
b	15	7	3	2	6
c	10	10	4	3	10
d	20	20	3	4	20

La ordenación de prioridades basada en el algoritmo de la cadencia monótona no puede asegurar una planificación del conjunto de tareas anterior, tal que se cumplan todos los tiempos límite de las tareas. Sin embargo, este conjunto de tareas nos saldría planificable si se asignan las prioridades con DMPO, tal como se hace en la 5ª columna de la tabla.

La demostración de la optimalidad de DMPO implica transformar las prioridades de cualquier esquema estático hasta que se obtiene la ordenación dada por DMPO. El esquema que se sigue en la demostración es el siguiente, aunque no se va a desarrollar completamente debido a su complejidad. Se parte de una ordenación de las prioridades a las tareas, dada, por ejemplo, por el algoritmo de cadencia monótona.

- Cada paso del procedimiento de transformación ha de preservar la planificabilidad de los procesos.
- Se intercambian tareas con prioridades adyacentes.

[9]DMPO es un criterio óptimo, ya que cualquier conjunto de tareas que sea planificable, atendiendo a un esquema de asignación de prioridades estático, también lo sería con este criterio.

- La tarea que disminuya su prioridad sigue siendo ejecutable.
- La tarea que aumenta su prioridad sólo interfiere a la otra.
- Como su tiempo límite es el menor de los dos, la otra tarea sigue siendo planificable después del intercambio.

Al final de proceso anterior se obtiene una ordenación de las tareas según la prioridad dada por el DMPO. Como durante la transformación del esquema inicial (cadencia monótona) al final (DMPO) no ha dejado de ser ninguna de las tareas planificable, se concluye que cualquier esquema de asignación de prioridades a las tareas estático es equivalente a DMPO, que se puede considerar óptimo entre estos.

4.3.2 Interacciones entre las tareas

Las entidades que constituyen un programa de tiempo real no son únicamente *tareas* que se ejecutan de forma asíncrona y sin interaccionar entre ellas, sino también existen *objetos protegidos*, es decir: secciones críticas, monitores, semáforos, etc., que proporcionan acceso en exclusión mutua a datos compartidos entre las tareas de la aplicación.

La utilización de objetos protegidos en los programas de tiempo real lleva a la posibilidad de que una tarea pueda ser bloqueada esperando a que se produzca algún evento futuro, distinto del de activación, que le permita continuar con su ejecución. Por ejemplo, una tarea podría quedar bloqueada esperando le ejecución de una operacion *signal*() de un semáforo por parte de otra tarea del programa. También, las tareas podrían entrar en una cola de un monitor hasta la ejecución de la operación *c.signal*(). O bien que la tarea realice una llamada a una cita, etc. Las *tareas* y los *objetos protegidos* pueden estar distribuidos a lo largo de un sistema físico con varios procesadores, por lo que se pueden producir esperas debidas a recepción de mensajes, acceso a buses, switches en multiprocesadores, etc.

El problema que surge si una tarea permanece bloqueada esperando a que otra menos prioritaria termine de ejecutar una sección de su código, es que deja de cumplirse una de las condiciones más importantes del modelo simple de tareas: *cuando una tarea tiene prioridad suficiente para ejecutarse, ha de hacerlo.* Según el modelo simple, en ningún caso podría verse bloqueada una tarea, o suspendida su ejecución durante algún tiempo, ya que esto influiría en la planificación del resto de las tareas y podría llegar a ocasionar la pérdida de sus tiempos límite. Por tanto, no sería de aplicación el análisis de planificabilidad basado en el Test de Liu-Layland, para el modelo simple de tareas, sin cuantificar cómo se vería afectada la desigualdad $\Sigma_{i=0}^{N} \frac{C_i}{T_i} \leq U_0(N)$ con el eventual bloqueo de las tareas al acceder a recursos comunes.

Inversión de prioridad

Podría suceder, incluso, que la espera de la tarea más prioritaria llegase a ser arbitrariamente larga si se ejecutan continuamente tareas menos prioritarias mientras que ésta se mantiene bloqueada. En ese caso, se produciría lo que se denomina una *inversión de prioridad* que

invalidaría cualquier previsión acerca de la planificabilidad de un conjunto de tareas (ver figura 4.8[10]). La inversión de prioridad no puede ser eliminada completamente si se utilizan objetos protegidos en los programas de tiempo real, pero los efectos adversos sobre la planificación de las tareas más prioritarias pueden ser minimizados, haciendo que el bloqueo derivado del acceso a dichos objetos sea siempre un tiempo acotado y medible en cualquier ejecución de la aplicación.

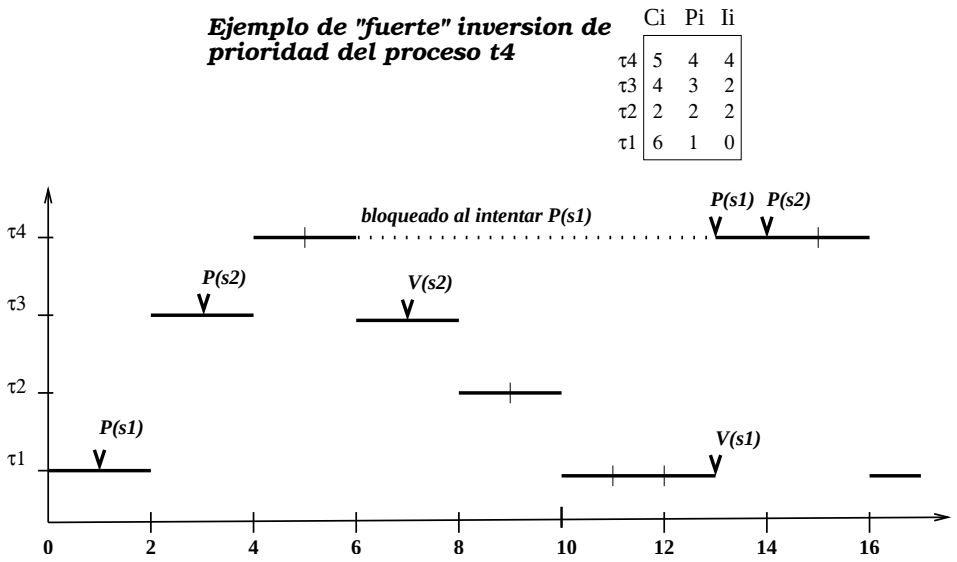


Figura 4.8: Inversión de prioridad de una tarea

Se puede decir que la *inversión de prioridad* es un inconveniente que se produce debido a un esquema estático de asignación de prioridades. En el ejemplo se puede observar como la tarea τ_4, la más prioritaria de todas, sufre una fuerte inversión de prioridad por parte de 3 tareas de menor prioridad:

La tarea τ_1 se activa primero, tras ejecutarse un *tick*[11], bloquea el cerrojo del semáforo s_1. Después resulta desplazada, cuando comienza la ejecución de la tarea τ_3, ya que ésta es más prioritaria que τ_1. Ésta última se ejecuta durante 1 *tick* y bloquea el cerrojo del semáforo s_2, después resulta desplazada por el comienzo de la tarea τ_4. La tarea τ_4 se ejecuta hasta que intenta adquirir el cerrojo del semáforo s_1, que lo tiene bloqueado τ_1, entonces τ_4 se bloquea a su vez. τ_3 vuelve a poseer el procesador, cuando termina de ejecutarse, τ_2 ejecuta las 2 unidades de tiempo que le quedan. Cuando ésta termina, puede volver a ejecutarse τ_1, hasta que libera el cerrojo de s_1, en ese momento, resulta ser desplazada por τ_4 que continúa su ejecución hasta terminar. Finalmente, la tarea τ_1 terminará de ejecutarse.

Sección crítica no expulsable

Es el protocolo más simple que se podría imaginar para evitar la inversión de prioridad en la implementación de un conjunto de tareas de tiempo real que comparten un recurso. Consiste en no permitir la expulsión del procesador de ninguna tarea cuando accede al recurso durante

[10] nótese que los números mayores indican más prioridad en la asignación del ejemplo de la figura
[11] unidad arbitraria de tiempo de ejecución

la ejecución de una sección crítica. El resultado de aplicar este protocolo a la ejecución de un conjunto de tareas es equivalente a la ejecución de las secciones críticas con una prioridad estática igual a la de la prioridad máxima del sistema.

Para aplicar este protocolo no se necesita tener un conocimiento previo de los requisitos asociados a los recursos. Sin embargo, puede inducir bloqueos de las tareas más prioritarias excesivamente largos si la duración del tiempo de ejecución de las secciones críticas no está acotado (o no puede acotarse antes de comenzar la ejecución del programa). Además, la puesta en práctica de este protocolo puede interferir en la ejecución de todas las tareas del sistema, aunque no hagan uso de los recursos compartidos. En la figura 4.9 puede verse un ejemplo[12] de aplicación del protocolo.

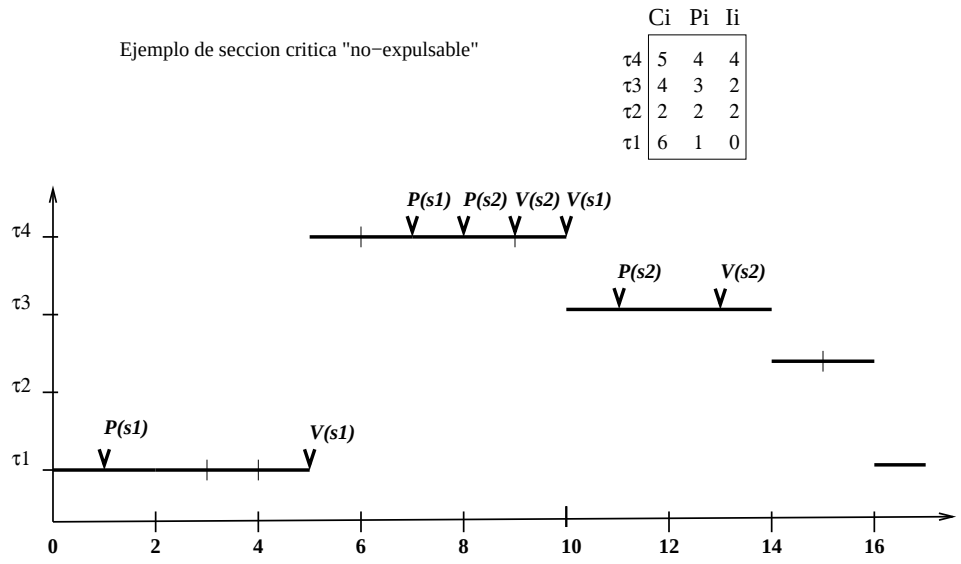


Figura 4.9: Escenario de tareas que usan "sección crítica no expulsable"

Tiempo de bloqueo

Se puede seguir utilizando el test de Liu-Layland si la inversión de prioridad representa un término constante en la desigualdad. En ese caso, se puede obtener, para cada tarea τ_i, la pérdida en la utilización del procesador que supone el que tareas menos prioritarias puedan bloquearle recursos que necesita mediante el cálculo del factor de bloqueo B_i. Luego, podemos aplicar el test de Liu–Layland para la tarea τ_i asumiendo que dicha tarea va a tener ahora un tiempo de ejecución de peor caso incrementado con el valor constante del factor de bloqueo calculado:

$$C_i^* = C_i + B_i$$

El factor de bloqueo se va a determinar considerando el peor caso posible de planificación, es decir, el mayor bloqueo que una tarea prioritaria podría experimentar debido a que comparte con varias tareas menos prioritarias que ella varias secciones críticas. Hay que tener en cuenta que cada protocolo que vamos a proponer para limitar el problema de la inversión de prioridad proporciona factores de bloqueo diferentes.

[12] nótese que los números mayores indican más prioridad en la asignación del ejemplo de la figura

Como con el protocolo de la sección crítica no expulsable una tarea prioritaria puede ser bloqueada como máximo durante la ejecución de una sección crítica. El factor de bloqueo vendrá dado por la siguiente expresión:

$$B_i = max_{j,\, j>i}(max_k(D(s_{jk}))$$

Donde s_{jk} es la sección crítica k ejecutada por la tarea τ_j menos prioritaria que τ_i y $D(s_{jk})$ es la duración de dicha sección crítica. La desigualdad $(j > i)$ referida a los índices de los identificadores de las tareas sirve para definir el conjunto de todas las tareas τ_j menos prioritarias que τ_i, pues suponemos que a las tareas más prioritarias se les asigna un índice menor.

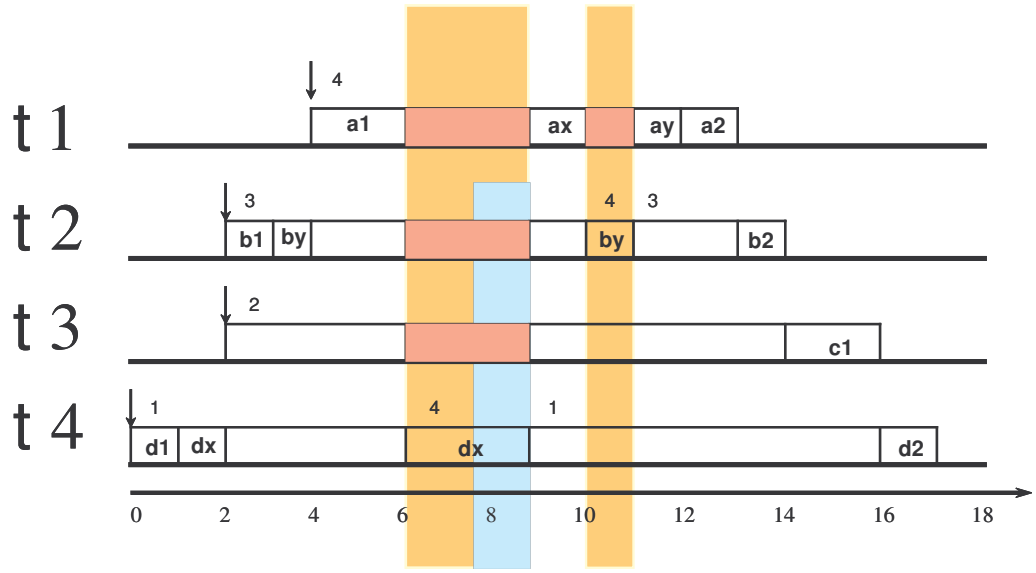


Figura 4.10: Escenario de 4 tareas con el protocolo de *herencia de prioridad*

Herencia de prioridad

Con este protocolo, las prioridades de las tareas no se mantienen estáticas durante toda la ejecución del programa y, como consecuencia de ello, se consigue minimizar el efecto de la *inversión de prioridad* en el aprovechamiento del tiempo del procesador por parte de las tareas.

Si la tarea más prioritaria τ_1 es bloqueada esperando que termine otra τ_2 de menor prioridad de ejecutar la sección crítica de un objeto protegido que comparte con la primera, entonces la prioridad de τ_2 se iguala a la de τ_1 durante el tiempo que τ_2 mantiene el cerrojo bloqueando a τ_1. Por tanto, durante su ejecución la prioridad de una tarea de la aplicación resultará ser el máximo entre su prioridad por defecto[13] y las prioridades de todas las demás tareas que en ese momento mantiene bloqueadas.

En la figura 4.10 se puede ver un escenario de ejecución de 4 tareas que acceden a 2 objetos protegidos, representados por las secciones críticas: $\{X,\ Y\}$. Los atributos de las tareas de la figura vienen dados en la siguiente tabla:

Las tareas pueden sufrir 2 tipos de bloqueos:

- Directo: lo sufre la τ_1 porque las tareas τ_4 y τ_2 la bloquean al acceder a las secciones críticas *dx* y *by*, que comparten con la primera.

[13]esto es, la prioridad con la que accede a la sección crítica

Tarea	P	C	Acciones
τ_1	1	5	a1(2);ax(1);ay(1);a2(1)
τ_2	2	4	b1(1);by(2);b2(1)
τ_3	3	2	c1(2)
τ_4	4	6	d1(1);dx(4);d2(1)

- Indirecto: las tareas τ_2 y τ_3 son bloqueadas por la tarea τ_4 por motivo de que la prioridad de ésta se ve elevada al valor máximo mientras está bloqueando a la tarea τ_1.

Con este protocolo las prioridades de las tareas cambiarán a menudo durante su ejecución, por lo que puede resultar ineficiente el implementarlo utilizando una cola de despacho de procesos ordenada por prioridad. Podría darse más de un bloqueo por inversión de prioridad durante la ejecución de una tarea prioritaria. Además, el protocolo de herencia de prioridad no evita ni el interbloqueo de las tareas en el acceso a recursos ni los bloqueos encadenados o *transitivos*.

Tiempo de bloqueo

La característica fundamental de este protocolo, con respecto a su influencia en el comportamiento dinámico de las tareas del programa, consiste en que las tareas sólo pueden verse bloqueadas un número limitado de veces por otras menos prioritarios. Con este protocolo, al elevarse la prioridad de la tarea que usa el objeto protegido a la de la tarea más prioritaria que esté bloqueando, ocasiona que la primera no pueda ser interrumpida por tareas de prioridad intermedia, que era la causa principal de la *inversión de prioridad*, por tanto:

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

El factor de bloqueo B_i en el caso del protocolo de herencia de prioridad vendrá dado como la sumatoria de todos los bloqueos que, en el peor de los escenarios de planificación posibles, podrían afectar a la tarea τ_i. Para calcular el factor de bloqueo de la tarea τ_i hay que establecer qué secciones críticas pueden estar en ejecución por otras tareas menos prioritarias cuando se active la primera y cuáles de éstas pueden bloquearla[14]. El factor de bloqueo para este protocolo suele ser difícil de calcular sistemáticamente y con exactitud. Por tanto, los algoritmos para hacerlo suelen proporcionar una estimación del mismo, que consiste en una cota superior del tiempo de bloqueo.

El factor de bloqueo vendrá dado como el mínimo entre 2 terminos a calcular,

1. B_i^l: bloqueo debido a tareas τ_j menos prioritarias, que acceden a secciones críticas k[15]:
   - $B_i^l = \Sigma_{j=i+1}^{n} \max_k[D_{j,k} : \text{Límite}(S_k) \geq P_i]$
2. B_i^s: bloqueo debido a todas las secciones críticas a las que accede la tarea τ_i,

[14] téngase en cuenta que el bloqueo aludido podría ser también del tipo *indirecto* –sin compartir sección crítica– por aumento de prioridad transitorio de las tareas

[15] aunque no necesariamente la comparten con la tarea τ_i, ya que su prioridad se podría elevar indirectamente.

- $B_i^s = \Sigma_{k=1}^{m} max_{j>i}[D_{j,k} : Límite(S_k) \geq P_i]$

Límite(S_k) es la prioridad de la tarea *cliente* más prioritaria de las que utilizan la sección crítica k. Sólo se consideran secciones que poseen un límite de prioridad no inferior a la prioridad de la tarea para la que se calcula el factor de bloqueo. $D_{j,k}$ es la duración de la ejecución de la sección crítica k por parte de la tarea τ_j.

Escribimos, por tanto, el factor de bloqueo de la tarea τ_i como: $B_i = Min(B_i^l, B_i^s)$.

Protocolos basados en el techo de prioridad

El protocolo de herencia de prioridad proporciona un límite superior del tiempo que puede permanecer bloqueada una tarea de alta prioridad. Sin embargo, se trata de una estimación del factor de bloqueo que puede resultar ser inaceptablemente pesimista, ya que, como se ha comentado anteriormente, el protocolo de herencia de prioridad no está libre de bloqueos *transitivos* o cadenas de bloqueos. Si se tiene en cuenta esta posibilidad en el cálculo del factor de bloqueo, el resultado será un valor límite superior excesivamente alto.

Los protocolos de techo de prioridad más utilizados en la actualidad son los siguientes:

1. protocolo original de límite de prioridad (OCPP),
2. protocolo inmediato de límite de prioridad (ICPP).

Justificaremos que, cuando se implementan utilizando un sistema monoprocesador, los protocolos de techo de prioridad cumplen lo siguiente:

- Una tarea prioritaria sólo puede ser bloqueada como máximo una vez durante su ejecución por otras de menor prioridad.
- Los interbloqueos se previenen.
- Se previenen los bloqueos transitivos.
- Se asegura el acceso en exclusión mutua a los recursos compartidos.

Con este tipo de protocolos se establece que las tareas pueden ser bloquedas en uno de estos dos casos: (a)cuando intentan bloquear un recurso previamente bloqueado por otra tarea; (b) si al bloquear un recurso se pudiera dar lugar a un bloqueo múltiple de tareas de mayor prioridad. Vamos a discutir a continuación cómo se puede conseguir obtener las propiedades anteriores para el protocolo ICPP, que es el contempla la norma POSIX para sistemas operativos.

El *techo de prioridad* ("priority ceiling") de un recurso es la prioridad de la tarea más prioritaria que puede bloquear a dicho recurso. Por lo tanto, el techo de prioridad de un recurso representa la mayor prioridad a la cual una sección crítica protegida puede ser ejecutada por una tarea de la aplicación.

Protocolo de techo de prioridad inmediato

Este protocolo es llamado en POSIX: *Priority Protected Protocol*(PPP). Se define mediante las siguientes reglas:

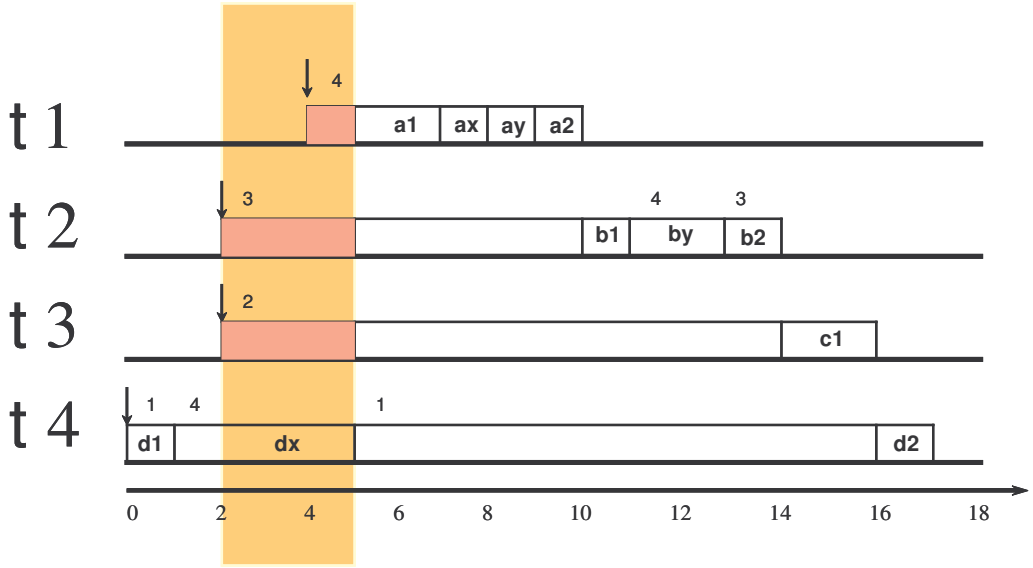


Figura 4.11: Escenario de 4 tareas con el protocolo *techo de prioridad inmediato*

1. Cada tarea tiene una prioridad estática asignada por defecto.

2. Cada recurso tiene un valor de techo de prioridad definido, que coincide con la máxima prioridad de las tareas que lo utilizan[16].

3. Una tarea tiene una prioridad dinámica que coincidirá con el máximo entre su propia prioridad estática inicial y los valores de los techos de prioridad de cualesquiera recursos que tenga bloqueados.

La tarea τ_4 de la figura 4.11 comienza a ejecutarse primero. Dado que el recurso X está libre cuando intenta acceder a él, adquiere el acceso al recurso y comienza a ejecutar la sección crítica elevándose su prioridad hasta el nivel máximo, que se corresponde con el techo de prioridad de dicho recurso. Hasta que la tarea τ_4 no termine de ejecutar las 4 unidades de tiempo de la sección crítica (dx) en ($t = 5$) ninguna de las otras tareas podrá comenzar a ejecutarse.

Las tareas sólo sufrirán bloqueo al principio de su ejecución. Ya que, cuando una tarea comienza, todos los recursos que vaya a utilizar han de estar libres, si no otras tareas tendrían una prioridad igual o mayor que dicha tarea (ya que en ese momento poseerían los recursos que la tarea pretende bloquear) y, por tanto, se vería bloqueada. En consecuencia, propiedades tales como la evitación de bloqueos transitivos y ausencia de interbloqueos son satisfechas por el protocolo.

El protocolo ICPP es más fácil de implementar que el protocolo de techo de prioridad original OCPP, pues produce menos cambios de contexto del planificador del sistema, ya que, como hemos comentado, el bloqueo de las tareas es previo a su ejecución. Por otra parte, la implementación del ICPP necesita, en general, más cambios de prioridad que el protocolo original, ya que estos se producen cada vez que una tarea bloquea recursos.

Los dos protocolos de límite de prioridad aseguran la exclusión mutua en el acceso a recursos compartidos, sin necesidad de usar primitivas de sincronización adicionales (semáforos, etc.). Ya que si una tarea obtiene acceso a algún recurso, entonces se ejecutará con el valor del límite

[16]Este valor se obtiene para recurso, observando el código de las tareas de la aplicación antes de la ejecución

de prioridad asignado a dicho recurso. Ninguna otra que utilice dicho recurso puede tener una prioridad mayor; por lo tanto, o bien la tarea accede al recurso sin ser interrumpida hasta termine, o la tarea resulta bloqueada antes de acceder a la sección crítica. En cualquier caso, la exclusión mutua en el acceso al recurso está asegurada con el protocolo ICPP mismo.

Tiempo de bloqueo

Dado que con los protocolos de techo de prioridad la tarea más prioritaria sólo puede sufrir un único bloqueo inicial, tal como se justificó anteriormente. Si utilizamos estos protocolos, entonces el valor máximo del tiempo de bloqueo B_i de una tarea τ_i es igual a la duración de la sección crítica más larga de las que acceden las tareas de prioridad inferior y que posean un límite de prioridad no menor que la prioridad de la primera ($prio(\tau_i)$).

Hay que tener en cuenta que una tarea puede verse bloqueada por otra menos prioritaria aunque no accedan a recursos comunes.

Por consiguiente, el factor de bloqueo para los protocolos de límite de prioridad vendrá dado como el resultado de evaluar la siguiente expresión:

$$B_i = Max_{\{j,k\}} \{D_{j,k} \mid prio(\tau_j) < prio(\tau_i),\ limite\_prioridad(S_k) \geq prio(\tau_i)\}$$

4.3.3 Algoritmos de planificación de tareas aperiódicas

Una forma simple de planificar tareas aperiódicas en un sistema de tiempo real sería la de asignarles una prioridad menor que la de las tareas cuya misión sea crítica. Sin embargo, actuando de esta manera, dichas tareas se ejecutarían sólo como actividades en *segundo plano*, e incumplirían frecuentemente sus tiempos límite. Por lo tanto, para mejorar la respuesta de las tareas que puedan perder ocasionalmente algún tiempo límite sin afectar a la seguridad del sistema[17] se puede emplear un *servidor* que consiste en una tarea real (o *conceptual*) que mantiene la planificación de los procesos cuya misión es crítica, pero, al mismo tiempo, permite que los procesos aperiódicos permisivos se puedan ejecutar tan pronto como sea posible.

Servicio de peticiones aperiódicas utilizando procesamiento de segundo plano

Esta solución, compatible con el test de Liu-Layland, consiste en asignar un tiempo extra para servir las peticiones de servicio de tareas aperiódicas. El denominado *tiempo aperiódico* se incrementa a intervalos regulares, siempre con la prioridad más baja del sistema, y se utiliza íntegramente para procesar las peticiones pendientes de activación tareas aperiódicas. Cuando no hay más peticiones aperiódicas pendientes se pierde, es decir, el tiempo aperiódico con este esquema no se preserva para atender las futuras peticiones aperiódicas que pudieran producirse.

[17] dichos sistemas los categorizamos como sistemas de tiempo real permisivos al inicio del tema

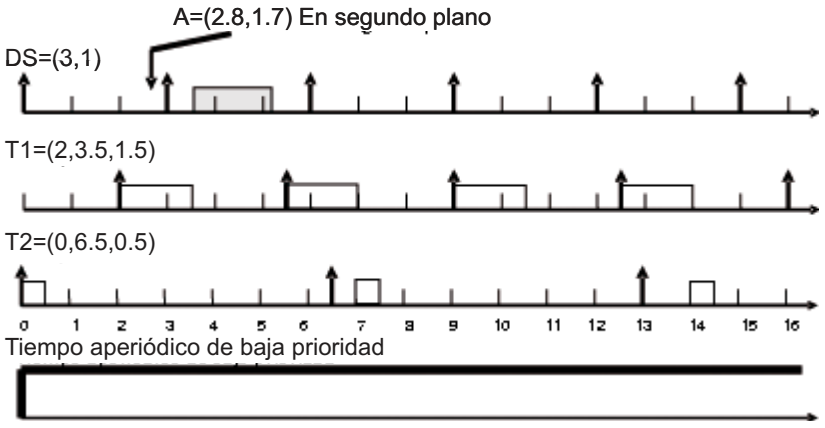


Figura 4.12: Escenario con peticiones aperiódicas en *segundo plano*

Servicio de peticiones aperiódicas utilizando sondeos

En este caso se mejoran un poco los tiempos de respuesta de las tareas aperiódicas respecto del método anterior. Se añade una tarea *sondeante* a las tareas periódicas de la aplicación, que puede ser una tarea física o conceptual. Es decir, no sería necesario crearla como tal, símplemente se puede reservar algún tiempo del procesador para sondear periódicamente si hay peticiones aperiódicas. Se trata de *tiempo periódico*, es decir, si no existen peticiones aperiódicas, se emplea en ejecutar las tareas periódicas activas. A diferencia del método anterior, se reserva un "tamaño" (C_s), equivalente al tiempo de ejecución de peor caso de una tarea periódica. La prioridad no se obtiene a partir del periodo (T_s), sino que se le puede asignar la prioridad que más convenga, para no provocar que las tareas periódicas pudieran perder algún límite de tiempo.

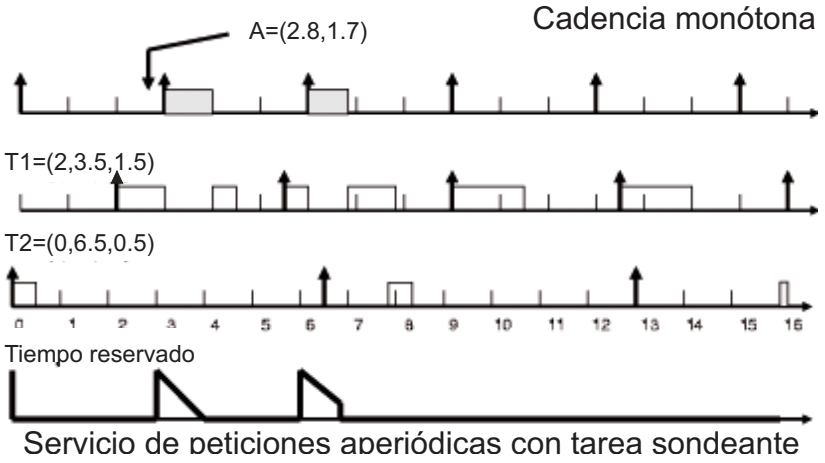


Figura 4.13: Escenario con peticiones aperiódicas y *tarea sondeante*

La planificabilidad de las tareas periódicas suele ser garantizada aplicando el test de Liu-Layland. Independientemente del número de tareas que pudieran solicitar servicio, en cada ciclo de sondeo se dedica un tiempo máximo igual a C_s para atender peticiones aperiódicas. La planificabilidad de un conjunto periódico con N tareas y asignación estática de prioridades según el algoritmo de cadencia monótona puede ser garantizado si y sólo si se cumple la desigualdad

siguiente:

$$\sum_{i=1}^{N} \frac{C_i}{T_i} \; + \; \frac{C_s}{T_s} \; \leq \; (N+1)[2^{\frac{1}{(N+1)}} \; - \; 1]$$

Algoritmos que preservan el tiempo asignado al procesamiento de peticiones aperiódicas.

Esta solución está basada en la implementación de un *servidor diferido*, que preserva el tiempo aperiódico, incluso si transitoriamente no hay peticiones de servicio de este tipo. Se asigna un tamaño al servidor (C_s) que se gasta en atender peticiones aperiódicas y se rellena hasta su valor máximo en cada ciclo del servidor (T_s). Se comienza realizando un análisis de planificabilidad del sistema de tiempo real, para determinar el *tamaño*[18](C_s) de una tarea servidora aperiódica de la máxima prioridad para incluirla en la ronda de planificación junto con el resto de las tareas periódicas.

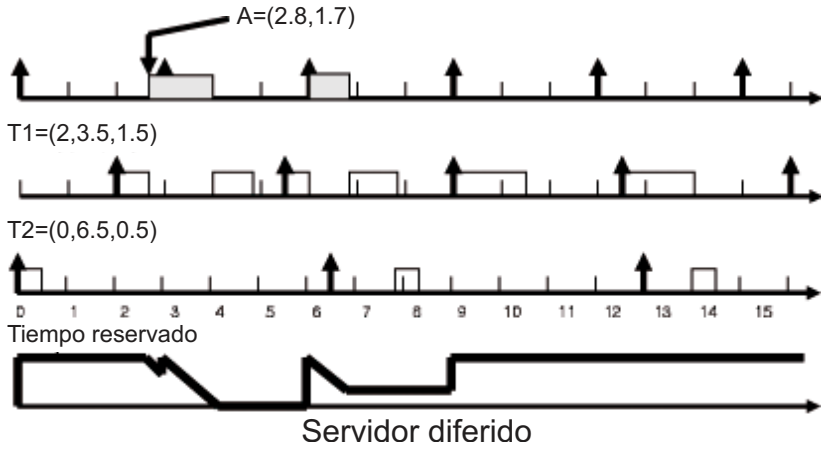


Figura 4.14: Escenario con peticiones aperiódicas y *servidor diferido*

Dichos valores se eligen de manera que todas las tareas clasificadas como de *misión crítica* del sistema mantengan siempre sus tiempos límite. De esta forma, se logra mantener el tiempo de ejecución asignado a las peticiones aperiódicas a un nivel de prioridad muy alto durante todo el ciclo de ejecución del servidor. Cuando sucede la activación de una tarea aperiódica se la sirve a la máxima prioridad, siempre que la capacidad del servidor no se haya agotado.

Análisis de planificabilidad

Considérese un conjunto de N tareas periódicas, $\tau_1 \ldots \tau_N$ y un *servidor diferido* con prioridad más alta. La condición de planificación en el peor caso se basa en un cálculo complejo, para obtener una función de la utilización del procesador por parte del servidor (U_s), es decir, se calcula el *menor límite superior de de utilización*[19]:

$$U_{mls} \; = \; U_s \; + \; N \left[\left(\frac{U_s \; + \; 2}{2U_s \; + \; 1} \right)^{\frac{1}{N}} \; - \; 1 \right]$$

[18]tiempo del procesador reservado para la ejecución de procesos aperiódicos

[19]es el mínimo de los límites de utilización entre los conjuntos de tareas que intentan usar todo el tiempo del procesador

Tomando el límite para $N \rightarrow \infty$, en el peor caso, encontramos como menor límite superior de utilización,

$$\lim_{N \rightarrow \infty} U_{mls} = U_s + \ln\left(\frac{U_s + 2}{2U_s + 1}\right)$$

Consecuentemente, dado un conjunto de N tareas periódicas y un *servidor diferido* con límites de utilización U_p y U_s, respectivamente, la planificabilidad del conjunto de tareas periódicas está garantizada, con el algoritmo de cadencia monótona, si $U_p + U_s \leq U_{mls}$; esto es, si se cunple la desigualdad siguiente:

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

4.4 Problemas resueltos

Ejercicio 1

Verificar la planificabilidad y construir el diagrama de ejecución de tareas utilizando el algoritmo de "cadencia monótona" (RM) para el siguiente conjunto de tareas periódicas

	C_i	T_i
τ_1	2	6
τ_2	2	8
τ_3	2	12

Solución:

Las tareas del conjunto anterior son planificables, ya que según el criterio RM, el factor de utilización del conjunto U $= \frac{2}{6} + \frac{2}{8} + \frac{2}{6} = 0.75$ es menor que la cota de utilización máxima RM: $U_0(3) = 3 \times (2^{\frac{1}{3}} - 1) \approx 0.78$. Tal como se puede ver en el diagrama de ejecución de las tareas de la figura 4.15.

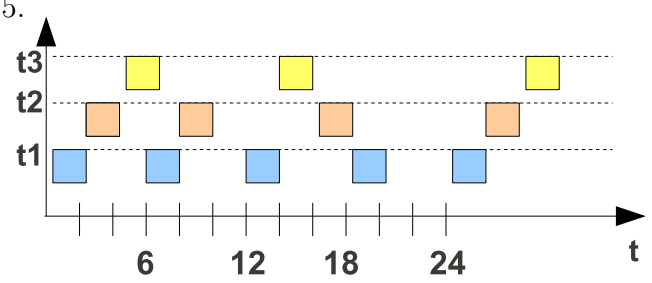


Figura 4.15: Diagrama de Gantt para las tareas del Ejercicio 1

Ejercicio 2

Verificar la planificabilidad y construir el diagrama de ejecución de tareas utilizando el algoritmo RM para el siguiente conjunto de tareas periódicas

	C_i	T_i
τ_1	3	5
τ_2	1	8
τ_3	2	10

Solución:

Con el test de planificabilidad RM no se puede afirmar que las tareas del conjunto anterior son planificables, ya que el factor de utilización del conjunto U $= \frac{3}{5} + \frac{1}{8} + \frac{2}{10} = 0.825$ es mayor que la cota de utilización máxima RM: $U_0(3)$. Sin embargo, se puede comprobar con el diagrama de ejecución de tareas de la figura 4.4 que son planificables.

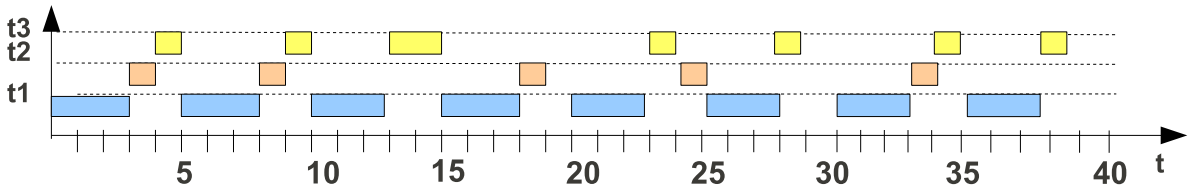


Figura 4.16: Diagrama de Gantt para las tareas del Ejercicio 2

Ejercicio 3

Verificar la planificabilidad del siguiente conjunto de tareas utilizando el algoritmo de "primero el plazo límite más cercano" (EDF)

	C_i	T_i
τ_1	1	4
τ_2	2	6
τ_3	3	10

Solución:

El conjunto de tareas es planificable con el algoritmo EDF si se toma el valor límite de D_i igual al del periodo de cada tarea ($D_i = T_i$), tal como se muestra en la figura 4.4. Se puede comprobar que factor de utilización del procesador que produce dicho conjunto es menor del 100%: $U = \frac{1}{4} + \frac{2}{6} + \frac{3}{8} = 0.96$

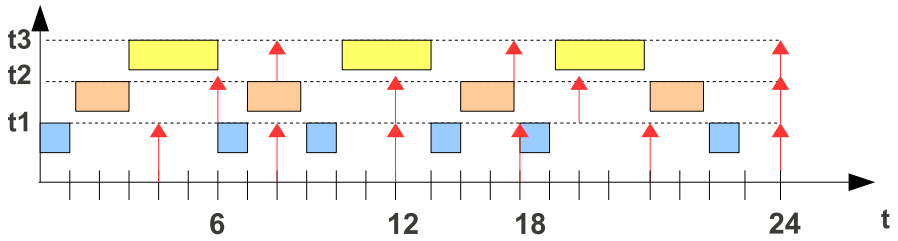


Figura 4.17: Diagrama de Gantt para las tareas del Ejercicio 3

Ejercicio 4

Verificar la planificabilidad utilizando el algoritmo EDF y construir el diagrama de ejecución de tareas del siguiente conjunto

	C_i	D_i	T_i
τ_1	2	5	6
τ_2	2	4	8
τ_3	4	8	12

Solución:

Este conjunto de tareas resulta ser planificable con el algoritmo EDF de asignación dinámica de prioridades a las tareas: $U_3 = \frac{2}{6} + \frac{2}{8} + \frac{4}{12} \approx 0.917 < 1.0$

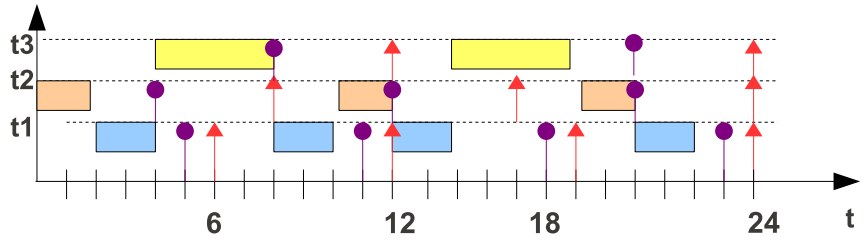


Figura 4.18: Diagrama de Gantt para las tareas del Ejercicio 4

Ejercicio 5

Verificar la planificabilidad del conjunto de tareas descrito en el Ejercicio 4, utilizando para ello el algoritmo del "plazo límite monótono" (*Deadline Monotonic* o DM)

Solución:

Este conjunto de tareas resulta no ser planificable con el algoritmo DM de asignación estática de prioridades a las tareas. Según se puede ver en la figura 4.4, la tarea τ_3 pierde su plazo de tiempo límite en t = $\{8, 20, \ldots\}$

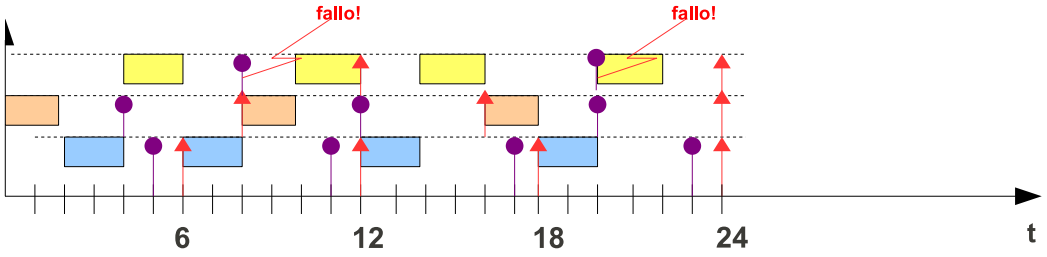


Figura 4.19: Diagrama de Gantt para las tareas del Ejercicio 5

Ejercicio 6

Calcular: a) la utilización máxima del procesador que se puede asignar al *Servidor Esporádico* para garantizar la planificabilidad del siguiente conjunto de tareas periódicas utilizando RM; b) la utilización del procesador máxima que puede ser asignada al Servidor Diferido (SD) para garantizar la planificabilidad del conjunto de tareas periódicas dado; c) un plan para planificar las siguientes tareas aperiódicas utilizando un *SE* que posea una utilización máxima y prioridad intermedia.

τ_1	1	5
τ_2	2	8

Solución (a):

El servidor esperádico (SE) se comporta como una tarea periódica. En la peor situación de planificación para el conjunto de tareas anterior, es decir, cuando exista la máxima interferencia entre ellas, el conjunto $\{\tau_1, \tau_2\}$ se puede garantizar como planificable con el algoritmo RM si se cumple la desigualdad $U_p \leq n \times ((\frac{2}{U_s+1})^{\frac{1}{n}} - 1) \Rightarrow U_s \leq 2 \times (\frac{U_p}{n} + 1)^n - 1$

Donde :

n	número de tareas periódicas
U_p	Utilización del procesador periódicas
U_s	Utilización del procesador aperiódicas

Para $n \to \infty$: $U_p \leq ln(\frac{2}{U_s+1})$ Luego, para n= 2: $U_{s\,max} = 0.33$ y $U_p = 0.45$.

Solución (b):

La utilización máxima U_s del procesador para el SD que garantice el mantenimiento de la planificabilidad del conjunto de tareas $\{\tau_1, \tau_2 \ldots\}$ viene dado por la siguiente inecuación:

$U_p \leq n \times ((\frac{U_s+2}{2 \cdot U_s+1})^{\frac{1}{n}} - 1)$
$\Rightarrow U_s \leq \frac{2-K}{2 \cdot K-1}$ donde $K = (\frac{U_p}{n+1})^n$

Para $n \to \infty$: $U_p \leq ln(\frac{U_s+2}{2 \cdot U_s+1})$ Luego, para n= 2: $U_{s\,max} = 0.25$ y $U_p = 0.45$.

Solución (c):

Sabemos que $U_{smax} = 0.33$ corresponde a la máxima utilización del procesador que podemos asignar al servidor para garantizar la planificabilidad del conjunto de tareas. Por tanto, asignando un periodo al servidor $T_s = 6$ (prioridad intermedia) y $C_s = 2$ se satisfacen las desigualdades y, por consiguiente, el conjunto de tareas $\{\tau_1, \tau_2\}$ se mantiene planificable aun con la aparición de las peticiones aperiódicas: $\{J_1, J_2, J_3\}$, según se puede ver en el siguiente diagrama de ejecución de tareas:

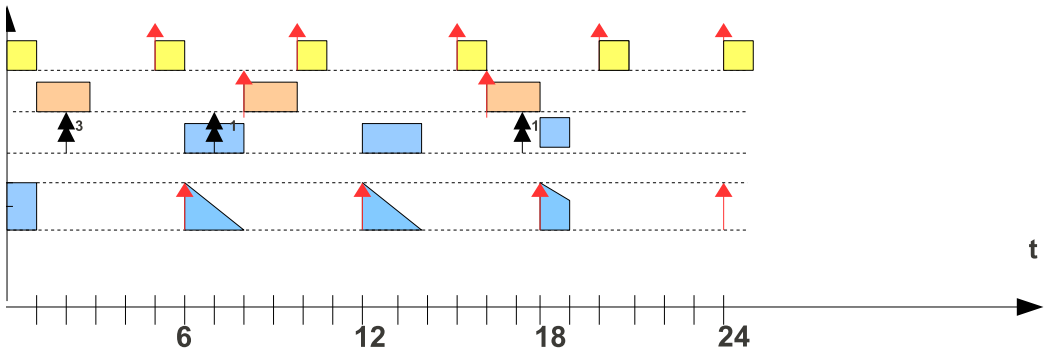


Figura 4.20: Diagrama de Gantt para las tareas del Ejercicio 6

4.5 Problemas propuestos

1. Verificar la planificabilidad (calcular el factor de utilización máxima y compararlo con U_3) y construir el diagrama de ejecución de tareas utilizando el algoritmo de "cadencia monótona" (RM) para el siguiente conjunto de tareas periódicas:

	C_i	T_i	Prio
τ_1	3	7	2
τ_2	3	12	2
τ_3	5	20	1

2. Verificar la planificabilidad y construir el diagrama de utilizando el algoritmo RM para el siguiente conjunto de tareas periódicas :

	C_i	T_i
τ_1	3	5
τ_2	1	8
τ_3	2	10

3. Verificar la planificabilidad y construir el diagrama de Gantt utilizando el algoritmo RM para el siguiente conjunto de tareas periódicas:

	C_i	T_i
τ_1	1	4
τ_2	2	6
τ_3	3	10

4. Calcular el *tiempo respuesta*(R) para cada una de las tareas del ejercicio anterior.

5. Verificar la planificabilidad y construir el diagrama de utilizando el algoritmo RM para el siguiente conjunto de tareas periódicas:

	C_i	T_i
τ_1	1	4
τ_2	2	6
τ_3	3	8

6. Calcular el *tiempo respuesta*(R) para cada una de las tareas del ejercicio anterior.

7. Calcular el tiempo de respuesta de cada una de las tareas de la tabla siguiente, calcular su tiempo de respuesta.

	C_i	T_i	Prio	D_i
τ_1	3	12	1	8
τ_2	6	20	2	10

8. Para el conjunto de tareas del ejercicio anterior, ¿Se puede encontrar una *implementación factible* para el conjunto de tareas, es decir, un programa de tiempo real asegurando que todas éstas terminarán cualquiera de sus activaciones antes que se cumplan los tiempos límite que tienen asignados en la tabla?

9. Para el conjunto de tareas $\{t_1,\ t_2,\ t_3\}$ cuyos datos se muestran más abajo, se pide:

   (a) Dibujar el diagrama de ejecución y obtener el tiempo de respuesta de cada una de las tareas.

   (b) Determinar, mediante inspección del gráfico, cuántas veces interfiere la tarea τ_1 a la tarea τ_3 durante el intervalo dado por el *tiempo de respuesta* de t_3.

   (c) Idem para las tareas τ_1 y τ_2.

   (d) Obtener el número máximo de veces que una tarea τ_j interfiere a otra tarea τ_i en función del periodo de la primera (T_j) y del tiempo de respuesta de la segunda (R_i).

	C_i	T_i	D_i
τ_1	1	3	2
τ_2	3	6	5
τ_3	2	13	13

10. Calcular el tiempo de finalización de la tarea τ_3 y determinar si las tareas A (servidor aperiódico), E (servidor esporádico) y las tareas periódicas: τ_1, τ_2 y τ_3 son todas ellas planificables.

	Tarea T_i	C_i	B(tiempo bloqueo acceso sscc)	Techo prio	P_i
A	40	2	0	-	2
τ_1	100	20	20	-	3
E	120	5	0	-	1
τ_2	150	40	20	-	4
τ_3	350	100	0	-	5
S_1	-	-	-	3	-
S_1	-	-	-	3	-

11. Utilizar un *Servidor Esporádico* con capacidad $C_s = 2$ y periodo $T_s = 5$ para planificar las siguientes tareas:

	C_i	T_i
τ_1	1	4
τ_2	2	6
	a_i	C_i
J_1	2	2
J_2	5	1
J_3	10	2

12. Para el conjunto de tareas cuyos datos se muestran más abajo, se pide:

   (a) Dibujar el gráfico de ejecución y obtener el tiempo de respuesta de cada tarea.

   (b) Determinar, mediante la inspección del gráfico anterior, cuántes veces interfiere la tarea τ_1 a la tarea τ_3 durante el intervalo temporal dado por el tiempo de respuesta de esta última tarea.

(c) Dibujar el diagrama de ejecución, obtener el tiempo de respuesta, e indicar el instante de tiempo en que cambia la prioridad dinámica de las tareas de la tabla siguiente. Suponemos que dichas tareas se planifican utilizando utilizando el protocolo denominado *herencia de prioridad*:

	C_i	T_i	I_i	Prio	recursos
τ_1	4	16	5	4	A, B
τ_2	4	16	5	3	-
τ_3	5	16	2	2	B,A
τ_4	5	18	0	1	A

13. Indicar si las afirmaciones siguientes son verdaderas:

(a) Suponiendo que las tareas se planifican con el protocolo de *herencia de prioridad*: la prioridad heredada por una tarea sólo se mantiene mientras dicha tarea esté utilizando un recurso compartido con otra tarea más prioritaria.

(b) Con el protocolo de *techo de prioridad*, cuando una tarea adquiere un recurso no puede verse interrumpida, hasta que termine su ejecución, por otras tareas que se activan después que ésta y que vayan a utilizar en el futuro un recurso con límite de prioridad igual o inferior.

(c) Con el protocolo de techo de prioridad (OCPPP), una tarea no puede comenzar a ejecutarse si no están libres todos los recursos que va a utilizar durante su primer ciclo.

(d) Si consideramos una tarea periódica que utilice el protocolo de techo de prioridad inmediato (ICPP) para cambiar su prioridad dinámica cuando accede a recursos, siempre se cumplirá que dicha tarea no puede ser interrumpida por otra menos prioritaria que ella.

(e) Con el protocolo de techo de prioridad las tareas más prioritarias del sistema pueden ser interrumpidas durante cada ciclo de su ejecución como máximo 1 vez cuando acceden a recursos que comparten con otras tareas menos prioritarias.

(f) El protocolo de techo de prioridad original (OCPP) producirá siempre tiempos de respuesta menores para las tareas que el algoritmo de *herencia de prioridad*.

14. Indicar si son ciertas las siguientes afirmaciones.

(a) Con asignación de prioridades basado en el menor tiempo límite relativo se obtiene un esquema de planificación dinámico (es decir, la prioridad de las tareas puede cambiar durante la ejecución del programa).

(b) Con asignación de prioridades según el algoritmo RM se puede asegurar la planificabilidad de un conjunto de n tareas periódicas si la utilización conjunta del procesador no supera el número $N \times (2^{\frac{1}{N}} - 1)$ siempre que todas las tareas comiencen al mismo tiempo.

(c) Si se utiliza asignación estática de prioridades a un conjunto de tareas periódicas independientes y todas acaban antes de que se cumpla el plazo de su tiempo límite en su primera activación, entonces podemos afirmar que siempre se cumplirán los plazos durante toda la ejecución de dichas tareas.

(d) Con un esquema de asignación de prioridades estático nunca se puede garantizar la planificabilidad de un conjunto de tareas periódicas independientes si la utilización conjunta del procesador $U_n = 100\%$.

(e) Suponiendo que el sistema utilice el algoritmo del *servidor diferido* para atender peticiones de servicio aperiódicas, afirmamos lo siguiente: si se agota la capacidad del servidor, es decir, el tiempo asignado para atender las tareas aperiódicas, entonces las dichas peticiones aperiódicas que pudieran llegar antes del comienzo de la siguiente reposición de tiempo del servidor se pierden.

(f) La tarea correspondiente al servidor diferido siempre se planifica con igual prioridad a la de la tarea periódica más prioritaria.

(g) El tiempo no utilizado del servidor diferido no se pierde, aunque no es acumulable.

(h) Suponiendo que se use el servicio de peticiones aperiódicas: si la capacidad del servidor aperiódico se ha terminado en el ciclo actual, se puede utilizar el tiempo sobrante asignado a las tareas periódicas.

(i) En cualquier caso, una petición aperiódica interrumpirá a cualquier tarea periódica activa en ese momento independientemente de la prioridad de esta última (suponer el algoritmo de intercambio de prioridad).

(j) Hay algoritmos de asignación estática de prioridades a las tareas en que la prioridad del servidor aperiódico puede cambiar con el tiempo.

(k) El tiempo que se haya consumido del servidor esporádico sólo se repone cuando se agota totalmente el tiempo del servidor y llega el siguiente ciclo de la tarea que implementa al servidor.

(l) Siempre que el nivel de prioridad del servidor esporádico vuelve a estar activo se repone el tiempo reservado para peticiones esporádicas.

(m) Podría ocurrir que la tarea que implementa el servidor esporádico fuese desplazada del procesador por una tarea periódica más prioritaria que ésta.

(n) Cuando se repone tiempo del servidor esporádico no ha de ser necesariamente hasta su nivel máximo.

(o) Suponiendo que las tareas se planifican con el protocolo de herencia de prioridad: la prioridad heredada por una tarea sólo se mantiene mientras utilice un recurso compartido con otra tarea más prioritaria.

(p) Con el protocolo de techo de prioridad, cuando una tarea adquiere un recurso no puede verse interrumpida, hasta que termine su ejecución, por otras tareas que vayan a utilizar un recurso de límite de prioridad igual o inferior.

(q) Con el protocolo de techo de prioridad, un proceso no puede comenzar a ejecutarse si no están libres todos los recursos que va a utilizar durante su primer ciclo.

15. Conceptos generales sobre STR. Seleccionar la respuesta correcta:

    (a) Los programas de tiempo real se caracterizan por la alta frecuencia de ejecución de las instrucciones de sus tareas.

    (b) Un sistema de tiempo real es *confiable* si sus procesos nunca se *cuelgan*.

    (c) Un sistema no puede ser de tiempo real si no se puede determinar el tiempo que necesitan todas sus tareas para terminar su primera activación.

    (d) La *responsividad* se refiere a la propiedad que tienen los sistemas de limitar el número de interrupciones anidadas que puede sufir cualquier tarea crítica.

16. Conceptos sobre medida del tiempo en los sistemas informáticos. Seleccionar la respuesta correcta:

    (a) Tiempo monónoto de un ordenador es el que nunca se agota y viene incluido en la fecha del sistema.

    (b) La *precisión* de un reloj de tiempo real se refiere a la unidad más pequeña de tiempo que se nos muestra en pantalla cuando ejecutamos la orden `gettimeofday(...)`.

    (c) Existen maneras de eliminar completamente la *deriva* en el tiempo fijado por un temporizador para el inicio de un tarea de tiempo real. En realidad sólo se trataría de descontar los retrasos externos a las tareas en cada ciclo de éstas.

    (d) Un computador puede tener tantos relojes de tiempo real POSIX como queramos programar.

17. Conceptos sobre el modelo simple de tareas de los STR. Seleccionar la respuesta correcta:

    (a) En este modelo las tareas nunca se pueden sincronizar.

    (b) El tiempo límite d_i de una tarea de tiempo real varía dependiendo del instante de activación.

    (c) El plazo de respuesta máximo D_i de una tarea depende del instante de tiempo en el que se produzca la siguiente activación de la tarea τ_i.

    (d) Si asignamos prioridades a las tareas según su menor plazo de respuesta, tenemos un esquema de prioridades estático; sin embargo, si consideramos la prioridad de una tarea mayor si su tiempo límite se encuentra más cercano, el esquema de prioridades será dinámico.

18. Conceptos generales sobre planificación de tareas periódicas. Seleccionar la respuesta correcta:

    (a) Con asignación de prioridades según el algoritmo RM se puede asegurar la planificabilidad de un conjunto de n tareas periódicas si la utilización conjunta del procesador no supera el número $N \times (2^{\frac{1}{N}} - 1)$ siempre que todas las tareas comiencen al mismo tiempo.

    (b) Si se utiliza asignación estática de prioridades a un conjunto de tareas periódicas independientes y todas acaban antes de que se cumpla el plazo de su tiempo límite en su primera activación, entonces podemos afirmar que siempre se cumplirán dichos plazos durante toda la ejecución de dichas tareas.

(c) Con un esquema de asignación de prioridades estático se podría tener conjuntos de tareas periódicas independientes y planificables llegando a la utilización conjunta del procesador $U_n = 100\%$.

(d) Con asignación de prioridades basado en el menor plazo de respuesta o tiempo límite relativo (D_i)se obtiene un esquema de planificación dinámico .

19. Servidores de tareas aperiódicas y esporádicas. Seleccionar la respuesta correcta:

(a) Si se agota la capacidad del *servidor diferido*, es decir, el tiempo asignado para atender las tareas aperiódicas, entonces las peticiones aperiódicas que pudieran llegar antes del comienzo de la siguiente reposición de tiempo del servidor se pierden.

(b) Suponiendo que se use el servicio de peticiones aperiódicas: si la capacidad del servidor aperiódico se ha terminado en el ciclo actual, se puede utilizar el tiempo sobrante asignado a las tareas periódicas.

(c) El tiempo no utilizado del servidor diferido no se pierde, aunque no es acumulable.

(d) En cualquier caso, una petición aperiódica interrumpirá a cualquier tarea periódica activa en ese momento independientemente de la criticidad de esta última.

20. Algoritmos para resolver el problema de la *inversión de prioridad*. Seleccionar la respuesta correcta:

(a) Suponiendo que las tareas se planifican con el protocolo de *herencia de prioridad*: la prioridad heredada por una tarea sólo se mantiene mientras dicha tarea esté utilizando un recurso compartido con otra tarea más prioritaria.

(b) Con el protocolo de *techo de prioridad inmediato*, una tarea no puede comenzar a ejecutarse si no están libres todos los recursos que va a utilizar durante su primer ciclo.

(c) Una tarea periódica que utilice el protocolo de techo de prioridad inmediato nunca puede ser interrumpida por otra menos prioritaria que ella.

(d) Con el protocolo de *herencia de prioridad* las tareas más prioritarias pueden ser interrumpidas durante cada ciclo de su ejecución como máximo 1 vez por otras tareas menos prioritarias.

Fuentes Consultadas

[Burns, 2003] Burns, A. (2003). *Sistemas de tiempo real y lenguajes de programación*. Addison-Wesley/Pearson Education, Madrid.

[Buttazzo, 2005] Buttazzo, G. (2005). *Hard real–time computing systems: predictable algorithms and applications*. Springer-Verlag, New-York.

[Cheng, 2002] Cheng, A. (2002). *Real-time systems: scheduling, analysis and verification*. Wiley, Hoboken, New Jersey.

[Gallmeister, 1995] Gallmeister, B. (1995). *Programming for the real world: POSIX 4.0*. O'Reilly, Sebastopol, California.

[Gomaa, 2000] Gomaa, H. (2000). *Designing concurrent, distributed and real–time applications with UML*. Addison-Wesley.

[Stankovic, 1988] Stankovic, J. (1988). Misconceptions about real-time computing: a serious problem for the next generation systems. *IEEE Computer*, 21(10):10:19.

[Wellings, 2004] Wellings, A. (2004). *Concurrent and real–time programming in Java*. John Wiley, N.J. (USA).

Lista de Figuras

Lista de Tablas

Bibliografía

[Agha, 1990] Agha, G. (1990). Concurrent object–oriented programming. *cacm*, 33(9):125–141.

[Agha et al., 1993] Agha, G., Mason, I., Smith, S., and Talcott, C. (1993). A foundation for actor computation. *Journal of Functional Computing*, 1(1):1–59.

[Andrews, 1991] Andrews, G. (1991). *Concurrent programming: principles and practice.* Benjamin Cummings, Redwood City, California.

[Andrews, 1999] Andrews, G. (1999). *Foundations of Multithreaded, Parallel, and Distributed Programming.* Benjamin Cummings, Redwood City, California.

[Arnold and Gosling, 2005] Arnold, K. and Gosling, J. (2005). *The Java Programming Language.* Professional. Addison-Wesley.

[Axford, 1989] Axford, T. (1989). *Concurrent Programming: Fundamnetal Techniques for Real–Time and Parallel Software Design.* John Wiley, Chichester (UK).

[Barnes, 1994] Barnes, J. (1994). *Programming in Ada. Plus an Overview of Ada 9X.* Addison-Wesley, New-York.

[Ben-Ari, 2006] Ben-Ari, M. (2006). *Principles of Concurrent and Distributed Programming.* 2nd Edition. Addison-Wesley.

[Birrel and Nelson, 1984] Birrel, A. and Nelson, B. (1984). Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59.

[Brinch-Hansen, 1975] Brinch-Hansen, P. (1975). The programming language concurrent pascal. *IEEE Transactions on Software Engineering*, 1(2):199–207.

[Brinch-Hansen, 1977] Brinch-Hansen, P. (1977). *The architecture of concurrent programs.* Prentice-Hall, Englewood Cliffs.

[Brinch-Hansen, 1995] Brinch-Hansen, P. (1995). *Parallel Programming Paradigms.* Studies in Computational Science. Prentice-Hall, Englewood Cliffs, NJ.

[Brinch-Hansen, 2002] Brinch-Hansen, P. (2002). *The Origin of Concurrent Programming: From Semaphores to Remote Procedures Calls.* Springer-Verlag, , New York.

[Burns, 2003] Burns, A. (2003). *Sistemas de tiempo real y lenguajes de programación.* Addison-Wesley/Pearson Education, Madrid.

[Butenhof, 1997] Butenhof, D. (1997). *Programming with POSIX threads*. Addison-Wesley.

[Buttazzo, 2005] Buttazzo, G. (2005). *Hard real–time computing systems: predictable algorithms and applications*. Springer-Verlag, New-York.

[Cheng, 2002] Cheng, A. (2002). *Real-time systems: scheduling, analysis and verification*. Wiley, Hoboken, New Jersey.

[Dahl et al., 1970] Dahl, O., Myhrhaug, B., and Nygaard, K. (1970). *SIMULA:Common Base Language*. Norwegian Computing Center, Oslo.

[Dijkstra, 1965] Dijkstra, E. (1965). Solution of a problem in concurrent programming control. *cacm*, 8(9):569.

[Dijkstra, 1971] Dijkstra, E. (1971). Hierarchical ordering of sequential processes. *Acta Informatica*, 1:115:138.

[Dijkstra, 1975] Dijkstra, E. (1975). Guarded commands, nondeterminacy, and formal derivation of programs. *cacm*, 18(8):453:457.

[Dillon, 1990] Dillon, L. (1990). Using symbolic execution for verification of ada tasking programs. *toplas*, 12(4):643–669.

[Eisenberg and M.R.McGuire, 1972] Eisenberg, M. and M.R.McGuire (1972). Further comments on dijkstra's concurrent programming control problem. *cacm*, 15(11):999.

[Gallmeister, 1995] Gallmeister, B. (1995). *Programming for the real world: POSIX 4.0*. O'Reilly, Sebastopol, California.

[Gehani, 1990] Gehani, N. (1990). Message passing in concurrent c: synchronous vs. asynchronous. *Software Practice and Experience*, 20(6):571–592.

[Gehani and Roome, 1986] Gehani, N. and Roome, W. (1986). Concurrent c. *Software Practice and Experience*, 16(9):821–844.

[Gehani and Roome, 1988] Gehani, N. and Roome, W. (1988). Rendez-voud facilities: Concurrent c and the ada language. *IEEE Transactions on Software Engineering*, 14(11):1546–1555.

[Gomaa, 2000] Gomaa, H. (2000). *Designing concurrent, distributed and real–time applications with UML*. Addison-Wesley.

[Hartley, 1998] Hartley, S. (1998). *Concurrent programming: thr Java programming language*. Oxford University Press, Oxford (UK).

[Hoare, 1985] Hoare, C. (1985). *Communicating sequential processes*. Prentice-Hall, London.

[Hoare, 1999] Hoare, C. (1999). Monitors: an operating system structuring concept. *cacm*, 10:594–557.

[Holt, 1983] Holt, R. (1983). *Concurrent Euclid, The UNIX System and Tunis*. Addison-Wesley, Reading, Massachusetts.

[Holt et al., 1987] Holt, R., Matthews, P., Roselet, J., and Cordy, J. (1987). *The TURING programming language: design and definition*. Prentice-Hall, Englewood Cliffs.

[Howard, 1976] Howard, L. (1976). Proving monitors. *cacm*, 19(5):273–278.

[INMOS, 1984] INMOS (1984). *Occam Programming Manual*. Prentice-Hall, USA.

[Knuth, 1966] Knuth, D. (1966). Additional comments on a problem in concurrent programming control. *cacm*, 9(5):321:322.

[Kumar et al., 2003] Kumar, V., Grama, A., Gupta, A., and Karypis, G. (2003). *Introduction to Parallel Computing*. Benjamin–Cummings.

[Lampson and Redell, 1980] Lampson, B. and Redell, D. (1980). Experience with processes and monitors in mesa. *cacm*, 23(2):105–117.

[Lea, 2001] Lea, D. (2001). *Programación concurrente en Java: principios y patrones de diseño*. Addison-Wesley/Pearson Education.

[Lister, 1977] Lister, A. (1977). The problem of nested monitor calls. *Operating Systems Review*, 11(3):5–7.

[Liu and Layland, 1973] Liu, C. and Layland, J. (1973). Scheduling algorithms for multiprogramming in a hard real-time environment. *jacm*, 20(1):46–61.

[Liu, 2004] Liu, M. (2004). *Computación distribuida: fundamentos y aplicaciones*. Pearson Education, Madrid.

[L.Lamport, 1974] L.Lamport (1974). A new solution of dijkstra's concurrent programming problem. *cacm*, 17(8):453–455.

[Meyer, 1993] Meyer, B. (1993). Systematic concurrent object–oriented programming. *cacm*, 36(9):37–46.

[Milner, 1980] Milner, R. (1980). A calculus of communicating systems. *LNCS*, 92.

[Peterson, 1983] Peterson, G. (1983). A new solution to lamport's concurrent programming problem using small shared variables. *toplas*, 5(1):56–65.

[Raynal, 1986] Raynal, M. (1986). *Algorithms for mutual exclusion*. North-Oxford Academic Publishers, London (UK).

[Raynal, 1988] Raynal, M. (1988). *Distributed algorithms and protocols*. John Wiley, Hoboken, N.J. (USA).

[Santoro, 2006] Santoro, N. (2006). *Design and Analysis of Distributed Algorithms*. John Wiley, N.J. (USA).

[Schneider and Andrews, 1986] Schneider, F. and Andrews, G. (1986). *Concepts for conurrent programming*, volume 224 of *Current trends in Concurrency–LNCS*. Springer-Verlag, N.Y. (USA).

[Scott, 1987] Scott, M. (1987). Language support for loosely coupled distributed programming. *IEEE Transactions on Software Engineering*, SE-13:88–103.

[Snir et al., 1999] Snir, M., Otto, S., Huss-Lederman, S., and D. Walker, J. D. (1999). *MPI: The Complete Reference*. The MIT Press, Cambridge, USA.

[Stankovic, 1988] Stankovic, J. (1988). Misconceptions about real-time computing: a serious problem for the next generation systems. *IEEE Computer*, 21(10):10:19.

[Udding, 1986] Udding, J. (1986). Absence of individual startvation using weak semaphores. *Information Processing Letters*, 23(3):159–162.

[Verissimo and Rodrigues, 2004] Verissimo, P. and Rodrigues, L. (2004). *Distributed systems for system architects*, volume I. Kluwer Academic, N.Y. (USA).

[von der Beeck, 2000] von der Beeck, M. (2000). Behaviour specifications: Equivalence and refinement notions. In *Visuelle Verhaltensmodellierung Verteilter und Nebenlaeufiger Software–Systeme, 8. Workshop des Arbeitskreises GROOM der GI Fachgruppe 2.1.9 OO Software-Entwicklung*, pages 1–5, Paderborn, D. Universitaet Munster.

[Wellings, 2004] Wellings, A. (2004). *Concurrent and real–time programming in Java*. John Wiley, N.J. (USA).

[Welsh and Bustard, 1979] Welsh, J. and Bustard, D. (1979). Pascal–plus: another language for modular multiprogramming. *Software Practice and Experience*, 9:947–957.

[Wirth, 1985] Wirth, N. (1985). *Programming in Modula-2*. Springer Verlag, Berlin.

[Wulf, 1969] Wulf, W. (1969). Performance monitors for multiprogramming systems. *2nd ACM Symposium on Operating Systems Principles*, pages 175–181.

Indice Alfabético

Appendix A

CSP

A.1 Introducción

La notación de programación que vamos a utilizar está basada en el CSP de C.A.R. Hoare y sus características fundamentales son:

1. Se proponen las *órdenes con guarda* de Dijkstra como estructuras de control secuenciales. Son la construcción fundamental para expresar el no-determinismo en el código de los procesos.

2. Una orden con una semántica parecida al *COBEGIN-COEND* para especificar la ejecución concurrente de los procesos, aunque con la restricción de que los procesos paralelos componentes no se pueden comunicar mediante variables globales

3. Ordenes especiales de entrada/salida entre los procesos:

   - `Pi` `Pj`
     `Pj!e` `Pi?x`

     - La comunicación tiene lugar cuando el proceso P_i nombra al proceso P_j como destino de su salida, y el proceso P_j nombra al proceso P_i como el origen de su entrada.
     - No existe almacenamiento intermedio (*buffering*) de los mensajes. Un proceso es retrasado en una orden de entrada/salida hasta que el otro proceso está preparado para la comunicación.

4. El paso de mensajes entre los procesos mantiene un control estricto de tipos, esto es, no se reciben mensajes cuyo tipo no concuerde con la declaración de tipo de la variable que ha de recibirlo. La declaración de tipos y variables es como en Pascal y se hace dentro de los procesos.

5. La notación de programación que vamos a utilizar es estática:

- El código del programa determina el número máximo de procesos que van a existir, esto es, el número de procesos se mantiene fijo durante toda la ejecución del programa, no se crean ni se destruyen procesos durante su ejecución.
- No existen llamadas recursivas a procedimientos o funciones

6. El lenguaje mantiene una sintaxis mínima. Las instrucciones del lenguaje que no están relacionadas con al comunicación entre procesos se sigue la notación del Pascal.

7. La notación de programación puede dar lugar a lenguajes implementables tanto en computadores convencionales (1 CPU), como en multicomputadores. No obstante, se necesitarán optimizaciones del código específicas para cada tipo de arquitectura.

La notación que se va a introducir se basa en *órdenes*. Una orden especifica el comportamiento del dispositivo que la ejecuta y puede tener éxito o fallar. Si la ejecución de una orden simple tiene éxito, entonces puede tener efecto en el estado interno del dispositivo que la ejecuta (*orden de asignación*), o en el entorno (*orden de salida*), o en ambos (*orden de entrada*).

La ejecución de una orden estructurada implica la ejecución de una o de todas las órdenes que la constituyen, dependiendo del tipo de orden.

BNF del modelo

```
            <orden>::= <orden.simple> | <orden.estructurada>
<orden.simple>::= <orden.nula> | <orden.asignacion> | <orden.entrada>
                        | <orden.salida>
<orden.estructurada>::= <orden.alternativa> | <orden.repetitiva>
                        | <orden.paralela> | <orden.con.guarda>
       <orden.nula>::= SKIP
    <lista.ordenes>::= {<declaracion>; | <orden>;} <orden>
```

La orden `SKIP` no tiene ningún efecto en el programa y nunca falla.

Una lista de órdenes especifica la ejecución secuencial de las órdenes que la constituyen, en el orden en que aparecen.

Cada declaración introduce una variable *fresca* cuyo ámbito se extiende desde el punto de su declaración hasta el final de la lista de órdenes.

A.2 Orden de asignación

Una orden de asignación `x:= e` especifica la evaluación de la expresión de la parte derecha y su asignación al valor denotado por la variable objetivo de la parte izquierda.

```
<orden.asignacion>::= <variable.objetivo>:= <expresion>
      <expresion>::= <expresion.simple> | <expresion.estructurada>
<variable.objetivo>::= <variable.simple> | <objetivo.estructurado>
```

Orden de evaluación
El valor denotado por la variable objetivo es evaluado después de tener éxito la asignación y es el mismo que el valor de la expresión evaluado antes de la asignación.

Asignación de listas de expresiones:
A una variable objetivo se le pueden asignar valores simples o estructurados:

```
<expresion.estructurada>::= <constructor>(<lista.expresion>)
<objetivo.estructurado>::= <constructor>(<lista.variables.objetivo>)
<constructor>::= <identificador> | <vacio>
<lista.expresion>::= <vacio> | <expresion>{, <expresion>}
<lista.variables.objetivo>::= <vacio> | <variable.objetivo>{,<variable.objetivo>}
```

Causas de fallo de una orden de asignación:

1. El valor de la expresión resulta indefinido, p.e. alguna de las operaciones que lo componen no se puede evaluar y resulta indefinida
2. El valor de la expresión no concuerda con la variable objetivo, para que concuerden: los constructores tienen que coincidir, la longitud de la lista de componentes de la variable objetivo ha de ser la misma que la lista de componentes de la expresión y los tipos individuales han de coincidir elemento a elemento.
   `x:= cons(izq, der)` se le asocia un valor estructurado a `x`.
   `P::[x:char; x:=7; cons(izq, der):= x]` falla.
   `x:= cons(izq, der)` construye un valor estructurado y se lo asigna a x.
   `insertar(n):= tiene(n)` falla por que no coinciden los constructores.

Una expresión estructurada con la lista de expresiones vacía es una señal. Lo mismo, en la variable objetivo, sirve para indicar que sólo se admite recibir ese tipo de señal.
`c:= P()`: asigna a `c` una señal con constructor `P` y sin componentes.
`P():= c`: falla si el valor de `c` no es `P()`; si lo es, no tiene efecto.

A.3 Ordenes de E/S

Las órdenes de E/S especifican la comunicación entre 2 procesos secuenciales en ejecución. Dicha comunicación tiene lugar si:

1. Una orden de entrada de un proceso Pa especifica como origen el nombre del otro proceso Pb
2. Una orden de salida del proceso Pb especifica como destino el proceso Pa
3. La variable objetivo de la orden de entrada concuerda con el valor denotado por la expresión de la orden de salida.

Si se cumplen las condiciones anteriores, entonces se dice que las órdenes de E/S se corresponden.

Las órdenes que se corresponden se ejecutan simultáneamente si no fallan y su efecto conjunto es asignar el valor de la expresión de la orden de salida a la variable objetivo de la orden de entrada correspondiente (concepto de *asignación remota*).
El requerimiento de sincronización entre las órdenes de entrada y salida, implica que se ha de retrasar al proceso que incluya a la orden que esté preparada primero para la comunicación. En cualquier caso, dicho retraso acaba cuando:

1. La orden correspondiente (en el otro proceso) está también preparada
2. El otro proceso termina y la orden del proceso que esperaba comunicación falla

Condiciones de fallo
Una orden de entrada falla, si el proceso que representa el origen del mensaje ha terminado.

$$[\mathrm{P1} \ :: \ \mathrm{P2}\,?\,\mathrm{x} \parallel \mathrm{P2} \ :: \ \mathrm{x} := 7]$$

Análogamente, una orden de salida falla, si el proceso que representa su destino ha terminado o si la expresión está indefinida.

Interbloqueos
Cuando un grupo de procesos está intentando comunicarse, pero ninguna de sus órdenes de E/S se corresponden; entonces, se dice que hay interbloqueo. Para que exista interbloqueo las órdenes de E/S de los procesos implicados no pueden fallar.

Ejemplo:

$$[\mathrm{P1} \ :: \ \mathrm{P2}\,!\,\mathrm{x} \parallel \mathrm{P2} \ :: \ \mathrm{P1}\,!\,\mathrm{y}]$$

Si la variable objetivo no concuerda con la orden de salida, entonces hay interbloqueo:

$$[\mathrm{P1} \ :: \ \mathrm{x} : \mathrm{char}; \ \mathrm{P2}\,?\,\mathrm{x} \parallel \mathrm{P2} \ :: \ \mathrm{P1}\,!\,(3,5)]$$

A.4 Orden paralela

Una orden paralela especifica la ejecución concurrente de sus procesos constituyentes. Todos ellos comienzan a la vez y la orden paralela termina con éxito sólo cuando todos hayan terminado (la velocidad relativa con que se ejecutan dichos procesos componentes es arbitraria).

```
 <orden.paralela> ::= [<proceso> { || <proceso>}]
         <proceso>::= <etiqueta.proceso><lista.ordenes>
<etiqueta.proceso>::= <vacio> | <identificador>:: | <identificador>(<subindice>)
                      {, <subindice>})::
       <subindice>::= <constante.entera> | <rango>
<constante.entera>::= <numero> | <variable.ligada>
 <variable.ligada>::= <identificador>
```

```
        <rango>::= <variable.ligada>:<cota.inferior>..<cota.superior>
<cota.inferior>::= <constante.entera>
<cota.superior>::= <constante.entera>
```

En una orden paralela, ningún proceso puede utilizar una variable que sea variable objetivo de cualquier otro proceso de la orden paralela.
Un proceso cuya etiqueta no tenga subíndices, o cuyos subíndices sean todos constantes, sirve para darle un nombre a la lista de órdenes que le sigue.
Un proceso cuyos subíndices de etiqueta definan 1 ó más *rangos* representan a una serie de procesos, cada uno con la misma etiqueta y lista de órdenes, excepto que cada uno de ellos tiene una combinación de valores distintos y que se sustituyen en las *variables ligadas*. Nota: variable ligada es aquella cuyos valores sólo pueden ser tomados dentro de una serie (normalmente un intervalo entero) que es especificado en el texto del programa; en el ejemplo siguiente, i sería una variable ligada. $X(i : 1..n) :: CL$ se expande a:

$$X(1) :: CL_1 \parallel X(2) :: CL_2 \parallel \ldots \parallel X(n) :: CL_n$$

.

A.5 Órdenes con guarda

Las guardas sirven para impedir la ejecución de las operaciones de comunicación entre los procesos, cuando no se dan las condiciones apropiadas para que se lleven a cabo (p.e. introducir datos en un buffer lleno, sacar datos de un buffer vacío).

Sintaxis de las órdenes con guarda

```
   <orden.con.guarda> ::=  <guarda> → <lista.ordenes>
                        | (<rango>{,<rango>})<guarda> → <lista.ordenes>
<guarda> ::= <lista.guardas> | <lista.guardas>;<orden.entrada> | <orden.entrada>
<lista.guardas> ::= <elemento.guarda> {;<elemento.guarda>}
<elemento.guarda> ::=  <expresion.booleana> | <declaracion>
```

Resultado de la ejecución de una orden con guarda
Una orden con guarda se ejecuta sii la ejecución de su guarda no falla. La ejecución de una orden con guarda tiene como resultado:

1. éxito: expresión booleana cierta y orden de paso de mensajes sin retraso,
2. fallo: expresión booleana falsa. También, si el proceso origen de la orden de entrada que aparece al final de la guarda ha terminado:
   `[P1::[P2 ? x -> SKIP]` ‖ `P2::SKIP]`, falla `P1`.
3. espera: la condición booleana es cierta y la orden de entrada se retrasa.

Orden de ejecución de las guardas. Ausencia de efectos laterales

Las guardas se ejecutan de izquierda a derecha.
El resultado de evaluar la expresión booleana de una guarda no tiene efecto en los valores de las variables locales del proceso (estado del proceso). Una posible implementación puede comprobar si una guarda falla, simplemente intentando ejecutarla y suspendiendo dicha ejecución en el momento en que falle. Lo anterior sería una implementación correcta, ya que suspender la ejecución de esta forma no tiene efecto en el estado del proceso.

La orden de entrada que aparece al final de una guarda se ejecuta sólo si la orden de salida correspondiente es también ejecutada.

A.6 Orden alternativa

Permite que los procesos seleccionen no-determinísticamente entre la ejecución de varias órdenes con guarda. El no-determinismo puede ser interpretado, a nivel de procesos, como la libertad de elección que tienen los procesos para elegir entre varias comunicaciones posibles. Dicha elección es realizada internamente por el proceso y no es controlada externamente por el entorno de dicho proceso (otros procesos que se ejecutan en paralelo con el).

Sintaxis de una orden alternativa:

```
<orden.alternativa> ::= [<orden.con.guarda> { □ <orden.con.guarda>}]
```

La ejecución de una orden alternativa consiste en la ejecución de 1 sóla de sus órdenes con guarda componentes.

Ausencia de equidad en la ejecución de las órdenes con guarda
No se asegura, en general, la equidad en la ejecución de la ordenes con guarda de una orden alternativa, incluso en el caso de que existan continuamente procesos preparados para comunicarse con las órdenes de entrada de las guardas anteriormente mencionadas. Esto es debido a que el no-determinismo no significa *aleatoriedad*. La interpretación correcta del sentido del

no-determinismo es que *el comportamiento deseado de los programas es invariante con respecto a la elección que realice un proceso que ejecute la orden alternativa.*

Si se observa un proceso no-determinista desde el exterior, no es posible predecir qué elección realizará el proceso en cada ocasión:

```
P1 ::  [(P2 ?  a → S1(a)) □ (P3 ?  b → S2(b))]
```

P1 tiene una elección: puede recibir en la variable `a` el valor recibido desde P2 y después comportarse como el proceso S1(a), o bien recibir el valor `b` y pasar a comportarse como el proceso S2(b). Si las 2 alternativas fueran posibles, P2 envía datos y P3 también, entonces P1 realiza una selección no-determinista no controlable por P2 o P3.

Ejemplo:

```
[(i:0..N-1) x:positive; contiene(i)= -1; in(i) ?  x -> contiene(i):= x]
```

El resultado de la ejecución de la orden anterior es que se introduce un nuevo elemento en el array `contiene`, suponiendo que todos los procesos `in(i)` esten preparados para enviar, pero no se puede saber cual. Aquellas órdenes con guarda que representan a elementos del array ya insertados no se tienen en cuenta.

Condición de fallo
Si todas las guardas fallan, entonces la orden alternativa falla, si no se selecciona arbitrariamente una de las órdenes con guarda que tenga éxito y se ejecuta, trás lo cual acaba la orden alternativa.

Cada guarda es ejecutada 1 sola vez por cada ejecución de la orden alternativa; por lo tanto, una guarda que falle no es tenida en cuenta en la ejecución de la orden alternativa.

Si las órdenes con guarda sufren retraso (con sus expresiones booleanas ciertas y esperando la comunicación de su orden de entrada) entonces la orden alternativa es retrasada.

```
[ (i:0..N-1) x:integer; in(i) ? x -> out ! (i, x) ]
```

Falla, si todos los procesos: `in(0), ... ,in(N-1)` han terminado.

A.7 Orden repetitiva

La orden repetitiva sirve para especificar tantas iteraciones como sean posibles de su orden alternativa constituyente.

La sintaxis de la orden repetitiva:

```
<orden.repetitiva>::= *<orden.alternativa>
```

Retraso en la ejecución de la orden repetitiva:
Si se tiene una orden repetitiva en que todas las guardas con éxito acaban en una orden de entrada que produce retraso, entonces la orden repetitiva se bloquea hasta que:

1. Existe una orden de salida preparada para enviar, correspondiente a una de las órdenes de entrada.
2. Todos los procesos origen nombrados en las órdenes de entrada de las guardas han terminado, entonces la orden repetitiva termina.
3. Hay interbloqueo si no se dan ninguna de las condiciones anteriores

Condición de terminacion

Cuando todas las guardas de su orden alternativa constituyente fallan, entonces la orden repetitiva termina sin ocasionar ningún efecto en su entorno; si no, la orden alternativa constituyente sería ejecutada ininterrumpidamente. Otro posible caso de terminación es que todos los procesos nombrados en las órdenes de entrada de las guardas hayan terminado.

Ejemplo1:

```
i:=0; *[i<tam; contenido(i)≠n -> i:= i+1]
```

Explora los elementos del array, para `i= 0, 1, 2, ...` hasta que `i` $\geq$ `tam` o se encuentra un valor igual a `n`, entonces la orden termina.

Ejemplo2:

```
mux:: *[(i:0..N-1) continua(i);x:integer;in(i) ? x ->
       out ! (i,x);continua(i):= FALSE]
```

Una posible terminación del ejemplo anterior se produciría si cada uno de los procesos `in(0),..., in(N-1)` hubiera terminado.

Condiciones para que existan interbloqueos

Para que exista un interbloqueo en un conjunto de procesos con órdenes repetitivas, no se puede cumplir alguna de las condiciones siguientes:

1. Algún par de órdenes de E/S se corresponden.
2. Todos los procesos nombrados en las órdenes de entrada de las guardas han terminado y como consecuencia de ello las ordenes repetitivas terminan.
3. Fallan todas las guardas en las órdenes repetitivas.

Ejemplos de interbloqueos:

```
[P1 ::  *[P2?x → S] ‖ P2::  P1?y] --no se cumplen (1)-(3)
                                  --y hay interbloqueo

[P1 ::  *[P2?x → S] ‖ P2::  P1!y] --se cumple (1),
                                  --no hay interbloqueo

[P1::  *[FALSE;P2?x → S] ‖ P2::  P1?y] --se cumple(3),
                                  --no hay interbloqueo

[P1::  [FALSE;P2 ?  x → S] ‖ P2::  P1 ?  y] --hay interbloqueo,
                                  --ya que no es una orden repetitiva
```

A.8 Ejemplos

1. El mayor de 2 números:

```
[(x ≥ y → m:= x ) □ (y ≥ x → m:= y)]
```

Si $x \geq y$, entonces asignar x a m, si no si $x \leq y$ entonces asignar y a m. Si ambos son posibles $x = y$, entonces se pueden ejecutar cualquiera de las 2 asignaciones nodeterminísticamente.

2. Los contadores de personas:

```
[P(j:1..2)::PERSONA || P(3)]

PERSONA::                          P(3)::
  *[i<20; P(3)!1 → i:= i+1]        cont:= 0;
                                   *[(j:1..2) P(j)?temp →  cont:= cont+temp)];
                                   WRITELN("numero de personas", cont)

          COBEGIN ||P1||P2||P3 COEND
```

Si la variable `i` alcanza el valor `20` en `P1, P2`, entonces `P3` termina.

3. Semáforo binario y semáforo general

```
*[Proc.usuario?P() →  Proc.usuario?V()]

*[X ? V() -> s:= s+1;
□ s>0; Y ? P() → s:= s-1]
```

Terminará cuando `X` e `Y` hayan terminado, o cuando `X` haya terminado y `s = 0`.

4. El productor/consumidor

```
BUFFER::                  productor::              consumidor::
b:array[0..N-1];          *[TRUE → generar.dato;   *[TRUE → BUFFER!algo();
in, out:  0..N-1;         BUFFER!dato]             BUFFER?x; consumir x]
tamaño:  0..N;
(in, out, tamaño):= (0, 0, 0);
*[(tamaño < N; productor?b[in] →
    tamaño:= tamaño + 1;
    in:= (in+1) MOD N)
  □ (tamaño > 0; consumidor?algo() →
    consumidor !  b[out];
    tamaño:= tamaño - 1;
    out:= (out + 1) MOD N)]
```

5. Cena de los filósofos

```
[filosofo(i:0..4)::filosofo || tenedor(i:0..4)::tenedor || habitacion]


tenedor::*[filosofo(i)?coger() -> filosofo(i)?soltar()
□ filosofo((i-1)MOD5)?coger() -> filosofo((i-1)MOD5)?soltar()]


filosofo::*[TRUE -> pensar
                    habitacion!entrar()
                    tenedor(i)!coger()
                    tenedor((i+1)MOD5)!coger()
                    comer
                    tenedor(i)!soltar()
                    tenedor((i+1)MOD5)!soltar()
                    habitacion!salir()]


habitacion::ocupacion:integer;ocupacion:=0; *[(i:0..4) ocupacion<4;
          filosofo(i)?entrar() -> ocupacion:= ocupacion+1
□(i:0..4) filosofo(i)?salir() -> ocupacion:= ocupacion-1]
```

6. Servidor de terminales.

$^*[(\texttt{i} : 1..10)\,\texttt{continue(i)};\ \texttt{console(i)?c} \rightarrow \texttt{x!(i,c)};\ \texttt{console(i)!ack};\ \texttt{continue(i)} := (\texttt{c} \neq \texttt{senial.termina}())]$. La orden recibe de cualquiera de las 10 consolas, dado que el elemento correspondiente del array booleano `continue` sea cierto. La variable ligada `i` identifica a la consola que origina el mensaje. `ack` es una señal de reconocimiento que se envía de vuelta a la consola que envía el mensaje. Si se recibe el carácter `senial.termina` se asigna `continue(i)` a falso, evitando posteriores envios desde dicha consola. La orden repetitiva termina cuando los 10 elementos de `continue` son falsos.