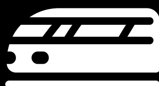
Programación Concurrente y Tiempo Real

Tercera Edición

Programación Concurrente y Tiempo Real

Tercera Edición

David Vallejo Fernández
Carlos González Morcillo
Javier A. Albusac Jiménez



Título: Programación Concurrente y Tiempo Real
Edición: 3ª Edición - Enero 2016
Autores: David Vallejo Fernández, Carlos González Morcillo y Javier A. Albusac Jiménez
ISBN: 978-1518608261
Edita: David Vallejo Fernández

Printed by CreateSpace, an Amazon.com company
Available from Amazon.com and other online stores

Este libro fue compuesto con LaTeX a partir de una plantilla de Carlos González Morcillo y David Villa Alises.

Creative Commons License: Usted es libre de copiar, distribuir y comunicar públicamente la obra, bajo las condiciones siguientes: 1. Reconocimiento. Debe reconocer los créditos de la obra de la manera especificada por el autor o el licenciador. 2. No comercial. No puede utilizar esta obra para fines comerciales. 3. Sin obras derivadas. No se puede alterar, transformar o generar una obra derivada a partir de esta obra. Más información en: http://creativecommons.org/licenses/by-nc-nd/3.0/

Prefacio

La evolución en el mercado de los procesadores y de los sistemas de procesamiento en general gira en torno a la integración de más unidades físicas de procesamiento que permitan ejecutar una mayor cantidad de tareas de manera simultánea. Una de las principales consecuencias de este planteamiento, desde el punto de vista de la programación, es la necesidad de utilizar herramientas que permitan gestionar adecuadamente el acceso concurrente a recursos y datos por parte de distintos procesos o hilos de ejecución, entre otros aspectos.

Este libro pretende ser una contribución, desde una perspectiva principalmente práctica, al diseño y desarrollo de sistemas concurrentes, haciendo especial hincapié en las herramientas que un programador puede utilizar para llevar a cabo dicha tarea. Así mismo, en este libro se introduce la importancia de estos aspectos en el ámbito de los sistemas de tiempo real.

Sobre este libro

Este libro discute los principales contenidos teóricos y prácticos de la asignatura *Programación Concurrente y Tiempo Real*, impartida en el segundo curso del *Grado en Ingeniería Informática* de la Escuela Superior de Informática de Ciudad Real (Universidad de Castilla-La Mancha). Puede obtener más información sobre la asignatura, código fuente de los ejemplos, prácticas de laboratorio y exámenes en la web de la misma: `http://www.libropctr.com`.

La versión electrónica de este libro puede descargarse desde la web anterior. La tercera edición, actualizada y revisada para corregir erratas, del libro «físico» puede adquirirse desde Amazon en `http://www.amazon.es/`.

Lectores destinatarios

Este libro está especialmente pensado para estudiantes de segundo o tercer curso de titulaciones de Ingeniería en Informática, como por ejemplo *Grado en Ingeniería Informática*. Se asume que el lector tiene conocimientos de Programación (particularmente en algún lenguaje de programación de sistemas como C) y Sistemas Operativos. El material que compone el presente libro ha sido desarrollado por profesores vinculados durante años a aspectos relacionados con la Programación, los Sistemas Operativos y los Sistemas Distribuidos.

Programas y código fuente

El código fuente de los ejemplos del libro puede descargarse en la siguiente página web: `http://www.libropctr.com`. Salvo que se especifique explícitamente otra licencia, todos los ejemplos del libro se distribuyen bajo GPLv3.

Agradecimientos

Los autores del libro quieren dar las gracias a los profesores Carlos Villarrubia Jiménez, Eduardo Domínguez Parra, Miguel Ángel Redondo Duque por su excepcional soporte en cualquier cuestión relacionada con el mundo de los Sistemas Operativos, particularmente en la generación de una imagen de sistema operativo especialmente modificada para la parte práctica de la asignatura *Programación Concurrente y Tiempo Real*. Este agradecimiento se hace extensivo al profesor Félix Jesús Villanueva Molina y Xerach Peña Ferrera por su soporte en la parte de Sistemas de Tiempo Real.

Gracias a Alba Baeza Pinés por la revisión gramatical y ortográfica del presente documento.

Finalmente, nuestro agradecimiento a la Escuela de Informática de Ciudad Real y al Departamento de Tecnologías y Sistemas de Información de la Universidad de Castilla-La Mancha.

Autores

David Vallejo (2009, Doctor Europeo en Informática, Universidad de Castilla-La Mancha) es Profesor Ayudante Doctor e imparte docencia en la Escuela de Informática de Ciudad Real (UCLM) en asignaturas relacionadas con Informática Gráfica, Programación y Sistemas Operativos desde 2007. Actualmente, su actividad investigadora gira en torno a la Vigilancia Inteligente, los Sistemas Multi-Agente y el Rendering Distribuido.

Carlos González (2007, Doctor Europeo en Informática, Universidad de Castilla-La Mancha) es Profesor Titular de Universidad e imparte docencia en la Escuela de Informática de Ciudad Real (UCLM) en asignaturas relacionadas con Informática Gráfica, Síntesis de Imagen Realista y Sistemas Operativos desde 2002. Actualmente, su actividad investigadora gira en torno a los Sistemas Multi-Agente, el Rendering Distribuido y la Realidad Aumentada.

Javier Albusac (2009, Doctor Europeo en Informática, Universidad de Castilla-La Mancha) es Profesor Ayudante Doctor e imparte docencia en la Escuela de Ingeniería Minera e Industrial de Almadén (EIMIA) en las asignaturas de Informática, Ofimática Aplicada a la Ingeniería y Sistemas de Comunicación en Edificios desde 2007. Actualmente, su actividad investigadora gira en torno a la Vigilancia Inteligente, Robótica Móvil y Aprendizaje Automático.

Resumen

Este libro recoge los aspectos fundamentales, desde una perspectiva esencialmente práctica y en el ámbito de los sistemas operativos, de *Programación Concurrente y de Tiempo Real*, asignatura obligatoria en el segundo curso del Grado en Ingeniería en Informática en la Escuela Superior de Informática (Universidad de Castilla-La Mancha). El principal objetivo que se pretende alcanzar es ofrecer al lector una visión general de las herramientas existentes para una adecuada programación de sistemas concurrentes y de los principales aspectos de la planificación de sistemas en tiempo real.

La evolución de los sistemas operativos modernos y el hardware de procesamiento, esencialmente multi-núcleo, hace especialmente relevante la adquisición de competencias relativas a la programación concurrente y a la sincronización y comunicación entre procesos, para incrementar la productividad de las aplicaciones desarrolladas. En este contexto, el presente libro discute herramientas clásicas, como los semáforos y las colas de mensajes, y alternativas más flexibles, como los monitores o los objetos protegidos. Desde el punto de vista de la implementación se hace uso de la familia de estándares POSIX y de los lenguajes de programación C, C++ y Ada.

Por otra parte, en este libro también se discuten los fundamentos de la planificación de sistemas en tiempo real con el objetivo de poner de manifiesto su importancia en el ámbito de los sistemas críticos. Conceptos como el *tiempo de respuesta* o el *deadline* de una tarea son esenciales para la programación de sistemas en tiempo real.

Abstract

This book addresses, from a practical point of view and within the context of Operating Systems, the basics of *Concurrent and Real-Time Programming*, a mandatory course taught in the second course of the studies in Computer Science at the Faculty of Computing (University of Castilla-La Mancha). The main goal of this book is to provide a general overview of the existing tools that can be used to tackle concurrent programming and to deal with real-time systems scheduling.

Both the evolution of operating systems and processing hardware, especially multicore processors, make the acquisition of concurrent programming-related competences essential in order to increase the performance of the developed applications. Within this context, this book describes in detail traditional tools, such as semaphores or message queues, and more flexible solutions, such as monitors or protected objects. From the implementation point of view, the POSIX family of standards and the programming languages C, C++ and Ada are employed.

On the other hand, this book also discusses the basics of real-time systems scheduling so that the reader can appreciate the importance of this topic and how it affects to the design of critical systems. Concepts such as *response time* and *deadline* are essential to understand real-time systems programming.

Índice general

1. Conceptos Básicos **1**

1.1. Concepto de Proceso 2

1.1.1. Gestión básica de procesos 2

1.1.2. Primitivas básicas en POSIX 3

1.1.3. Procesos e hilos 11

1.2. Fundamentos de P. Concurrente 14

1.2.1. El problema del productor/consumidor 15

1.2.2. La sección crítica 15

1.2.3. Mecanismos básicos de sincronización 21

1.2.4. Interbloqueos y tratamiento del *deadlock* 27

1.3. Fundamentos de T. Real 36

1.3.1. ¿Qué es un sistema de tiempo real? 37

1.3.2. Herramientas para sistemas de tiempo real 39

2. Semáforos y Memoria Compartida **41**

2.1. Conceptos Básicos 42

2.2. Implementación 45

2.2.1. Semáforos 45

2.2.2. Memoria compartida 47

2.3. Problemas Clásicos 50

2.3.1. El buffer limitado 50

2.3.2. Lectores y escritores 53

2.3.3. Los filósofos comensales 57

2.3.4. El puente de un solo carril 61
2.3.5. El barbero dormilón 63
2.3.6. Los caníbales comensales 66
2.3.7. El problema de Santa Claus 68

3. Paso de Mensajes **73**
3.1. Conceptos Básicos 73
3.1.1. El concepto de buzón o cola de mensajes 75
3.1.2. Aspectos de diseño en sistemas de mensajes 75
3.1.3. El problema de la sección crítica 78
3.2. Implementación 79
3.2.1. El problema del bloqueo 85
3.3. Problemas Clásicos 87
3.3.1. El buffer limitado 88
3.3.2. Los filósofos comensales 88
3.3.3. Los fumadores de cigarrillos 90
3.3.4. Simulación de un sistema de codificación 92

4. Otros Mecanismos de Sincronización **95**
4.1. Motivación 96
4.2. Concurrencia en Ada 95 97
4.2.1. Tareas y sincronización básica 98
4.2.2. Los objetos protegidos 103
4.3. El concepto de monitor 109
4.3.1. Monitores en POSIX 111
4.4. Consideraciones finales 116

5. Planificación en Sistemas de Tiempo Real **117**
5.1. Introducción 117
5.2. El concepto de *tiempo real* 118
5.2.1. Mecanismos de representación 119
5.2.2. Control de requisitos temporales 124
5.3. Esquemas de planificación 126
5.3.1. Modelo simple de tareas 128
5.3.2. El ejecutivo cíclico 128
5.3.3. Planificación basada en procesos 131

5.4. Aspectos relevantes de un planificador 132
5.4.1. Sistema de asignación de prioridades 132
5.4.2. Modelo de análisis del tiempo de respuesta 133
5.4.3. Extendiendo el modelo simple de tareas 139
5.4.4. Consideraciones finales 150

A. El puente de un solo carril **151**
A.1. Enunciado 151
A.2. Código fuente 152

B. Filósofos comensales **159**
B.1. Enunciado 159
B.2. Código fuente 160

C. La biblioteca de hilos de ICE **165**
C.1. Fundamentos básicos 165
C.2. Manejo de hilos 166
C.3. Exclusión mutua básica 170
C.3.1. Evitando interbloqueos 172
C.3.2. Flexibilizando el concepto de *mutex* 174
C.4. Introduciendo monitores 174
C.4.1. Ejemplo de uso de monitores 176

Capítulo 1

Conceptos Básicos

En este capítulo se plantean los aspectos básicos relativos al **concepto de proceso**, estableciendo las principales diferencias con respecto a una hebra o hilo y haciendo especial hincapié en su creación y gestión mediante primitivas POSIX. Así mismo, también se establece un marco general para el estudio de los fundamentos de programación concurrente. Estos aspectos se discutirán con más detalle en sucesivos temas.

La problemática que se pretende abordar mediante el estudio de la programación concurrente está vinculada al concepto de **sistema operativo multiproceso**, donde los procesos comparten todo tipo de recursos, desde la CPU hasta una impresora. Este planteamiento mejora la eficiencia del sistema, pero plantea la cuestión de la **sincronización** en el acceso a los recursos. Típicamente, esta problemática no se resuelve a nivel de sistema operativo, siendo responsabilidad del programador el garantizar un acceso consistente a los recursos.

Comunicando procesos

Los procesos necesitan algún tipo de mecanismo explícito tanto para compartir información como para sincronizarse. El hecho de que se ejecuten en una misma máquina física no implica que los recursos se compartan de manera implícita sin problemas.

Este planteamiento general se introduce mediante el problema clásico del productor/consumidor, haciendo hincapié en la necesidad de compartir un buffer, y da lugar al concepto de **sección crítica**. Entre las soluciones planteadas, destaca el uso de los **semáforos** y el **paso de mensajes** como mecanismos clásicos de sincronización.

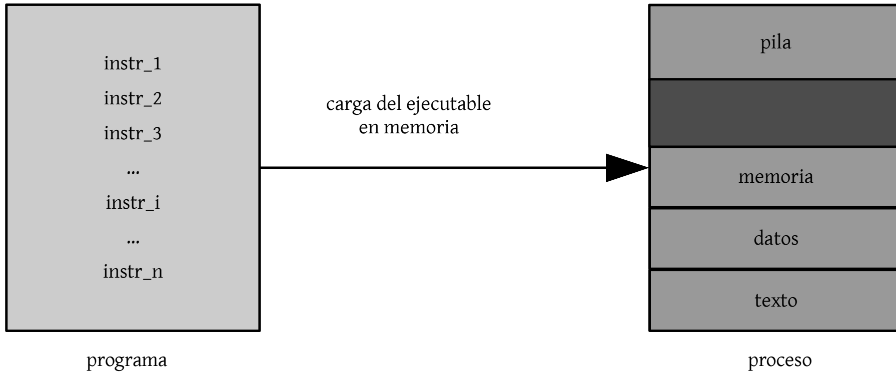


Figura 1.1: Esquema gráfico de un programa y un proceso.

1.1. El concepto de proceso

Informalmente, un proceso se puede definir como un **programa en ejecución**. Además del propio código al que está vinculado, un proceso incluye el valor de un contador de programa y el contenido de ciertos registros del procesador. Generalmente, un proceso también incluye la pila del proceso, utilizada para almacenar datos temporales, como variables locales, direcciones de retorno y parámetros de funciones, y una sección de datos con variables globales. Finalmente, un proceso también puede tener una sección de memoria reservada de manera dinámica. La figura 1.1 muestra la estructura general de un proceso.

1.1.1. Gestión básica de procesos

A medida que un proceso se ejecuta, éste va cambiando de un estado a otro. Cada **estado** se define en base a la actividad desarrollada por un proceso en un instante de tiempo determinado. Un proceso puede estar en uno de los siguientes estados (ver figura 1.2):

- **Nuevo**, donde el proceso está siendo creado.
- **En ejecución**, donde el proceso está ejecutando operaciones o instrucciones.
- **En espera**, donde el proceso está a la espera de que se produzca un determinado evento, típicamente la finalización de una operación de E/S.
- **Preparado**, donde el proceso está a la espera de que le asignen alguna unidad de procesamiento.
- **Terminado**, donde el proceso ya ha finalizado su ejecución.

En UNIX, los procesos se identifican mediante un entero único denominado **ID del proceso**. El proceso encargado de ejecutar la solicitud para crear un proceso se denomina *proceso padre*, mientras que el nuevo proceso se denomina *proceso hijo*. Las primitivas POSIX utilizadas para obtener dichos identificadores son las siguientes:

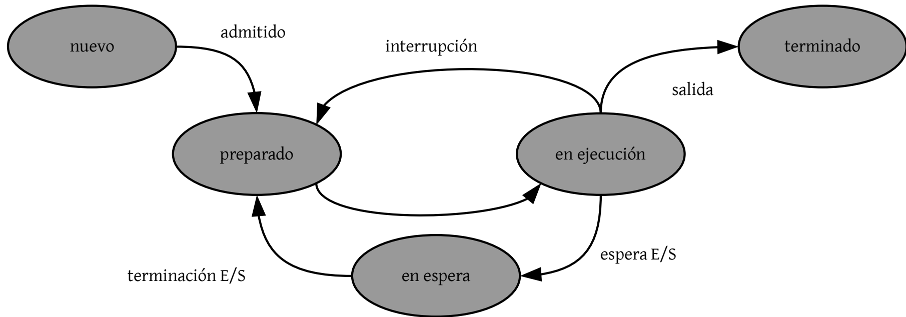


Figura 1.2: Diagrama general de los estados de un proceso.

Listado 1.1: Primitivas POSIX ID Proceso

```
1 #include <sys/types.h>
2 #include <unistd.h>
3
4 pid_t getpid  (void); // ID proceso.
5 pid_t getppid (void); // ID proceso padre.
6
7 uid_t getuid  (void); // ID usuario.
8 uid_t geteuid (void); // ID usuario efectivo.
```

Las primitivas *getpid()* y *getppid()* se utilizan para obtener los identificadores de un proceso, ya sea el suyo propio o el de su padre.

Proceso init

En sistemas UNIX y tipo UNIX, *init* (*initialization*) es el primer proceso en ejecución, con PID 1, y es el responsable de la creación del resto de procesos.

Recuerde que UNIX asocia cada proceso con un usuario en particular, comúnmente denominado *propietario del proceso*. Al igual que ocurre con los procesos, cada usuario tiene asociado un ID único dentro del sistema, conocido como **ID del usuario**. Para obtener este ID se hace uso de la primitiva *getuid()*. Debido a que el ID de un usuario puede cambiar con la ejecución de un proceso, es posible utilizar la primitiva *geteuid()* para obtener el ID del *usuario efectivo*.

1.1.2. Primitivas básicas en POSIX

La creación de procesos en UNIX se realiza mediante la llamada al sistema **fork()**. Básicamente, cuando un proceso realiza una llamada a *fork()* se genera una **copia exacta** que deriva en un nuevo proceso, el *proceso hijo*, que recibe una copia del espacio de direcciones del proceso padre. A partir de ese momento, ambos procesos continúan su ejecución en la instrucción que está justo después de *fork()*. La figura 1.3 muestra de manera gráfica las implicaciones derivadas de la ejecución de *fork()* para la creación de un nuevo proceso.

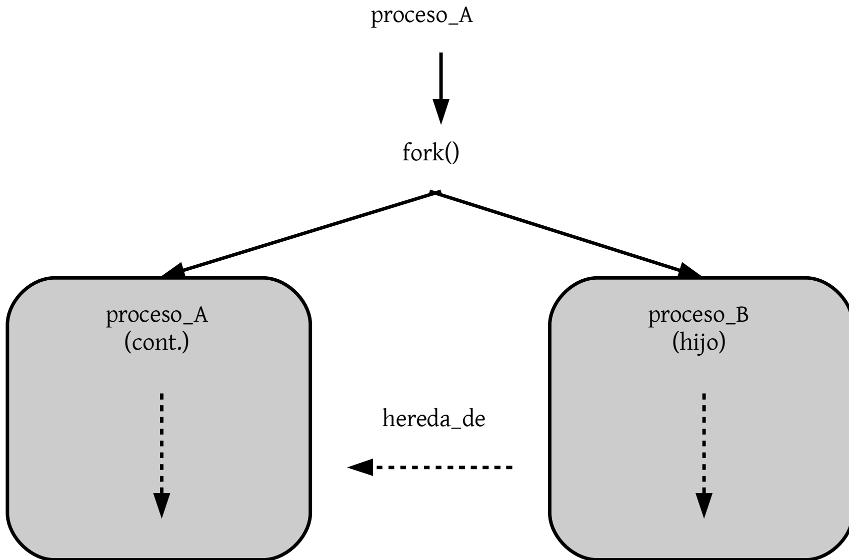


Figura 1.3: Creación de un proceso mediante fork().

Listado 1.2: La llamada fork() al sistema

```
1 #include <sys/types.h>
2 #include <unistd.h>
3
4 pid_t fork (void);
```

Independencia

Los procesos son independientes entre sí, por lo que no tienen mecanismos implícitos para compartir información y sincronizarse. Incluso con la llamada *fork()*, los procesos padre e hijo son totalmente independientes.

Recuerde que *fork()* devuelve el valor 0 al hijo y el PID del hijo al padre. Dicho valor permite distinguir entre el código del proceso padre y del hijo, para asignar un nuevo fragmento de código. En otro caso, la creación de dos procesos totalmente idénticos no sería muy útil.

fork() devuelve 0 al proceso hijo y el PID del hijo al padre.

El listado de código 1.3 muestra un ejemplo muy básico de utilización de *fork()* para la creación de un nuevo proceso. No olvide que el proceso hijo recibe una copia exacta del espacio de direcciones del proceso padre.

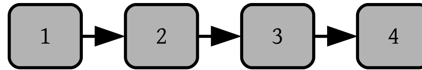


Figura 1.4: Cadena de procesos generada por el listado de código anexo.

Listado 1.3: Uso básico de fork()

```
1  #include <sys/types.h>
2  #include <unistd.h>
3  #include <stdlib.h>
4  #include <stdio.h>
5
6  int main (void) {
7    int *valor = malloc(sizeof(int));
8    *valor = 0;
9    fork();
10   *valor = 13;
11   printf("%ld: %d\n", (long)getpid(), *valor);
12
13   free(valor);
14   return 0;
15 }
```

La salida de este programa sería la siguiente[1]:

```
13243: 13
13244: 13
```

¿Cómo distinguiría el código del proceso hijo y el del padre?

Después de la ejecución de *fork()*, existen dos procesos y cada uno de ellos mantiene una copia de la variable *valor*. Antes de su ejecución, solamente existía un proceso y una única copia de dicha variable. Note que no es posible distinguir entre el proceso padre y el hijo, ya que no se controló el valor devuelto por *fork()*.

Típicamente será necesario crear un número arbitrario de procesos, por lo que el uso de *fork()* estará ligado al de algún tipo de estructura de control, como por ejemplo un bucle *for*. Por ejemplo, el siguiente listado de código genera una cadena de procesos, tal y como se refleja de manera gráfica en la figura 1.4. Para ello, es necesario asegurarse de que el proceso generado por una llamada a *fork()*, es decir, el proceso hijo, sea el responsable de crear un nuevo proceso.

Anteriormente se comentó que era necesario utilizar el valor devuelto por *fork()* para poder asignar un código distinto al proceso padre y al hijo, respectivamente, después de realizar dicha llamada. El uso de *fork()* por sí solo es muy poco flexible y generaría una gran cantidad de código duplicado y de estructuras *if-then-else* para distinguir entre el proceso padre y el hijo.

[1]El valor de los ID de los procesos puede variar de una ejecución a otra.

Idealmente, sería necesario algún tipo de mecanismo que posibilitara asignar un nuevo módulo ejecutable a un proceso después de su creación. Este mecanismo es precisamente el esquema en el que se basa la familia **exec** de llamadas al sistema. Para ello, la forma de combinar *fork()* y alguna primitiva de la familia *exec* se basa en permitir que el proceso hijo ejecute *exec* para el nuevo código, mientras que el padre continúa con su flujo de ejecución normal. La figura 1.5 muestra de manera gráfica dicho esquema.

Listado 1.4: Creación de una cadena de procesos

```
1  #include <sys/types.h>
2  #include <unistd.h>
3  #include <stdio.h>
4
5  int main (void) {
6    int i = 1, n = 4;
7    pid_t childpid;
8
9    for (i = 1; i < n; i++) {
10     childpid = fork();
11     if (childpid > 0) { // Código del padre.
12       break;
13     }
14   }
15
16   // ¿PID == 1?
17   printf("Proceso %ld con padre %ld\n", (long)getpid(), (long)getppid());
18
19   pause();
20
21   return 0;
22 }
```

El uso de operaciones del tipo *exec* implica que el proceso padre tenga que integrar algún tipo de mecanismo de espera para la correcta finalización de los procesos que creó con anterioridad, además de llevar a cabo algún tipo de liberación de recursos. Esta problemática se discutirá más adelante. Antes, se estudiará un ejemplo concreto y se comentarán las principales diferencias existentes entre las operaciones de la familia *exec*, las cuales se muestran en el listado de código 1.5.

Listado 1.5: Familia de llamadas exec

```
1  #include <unistd.h>
2
3  int execl  (const char *path, const char *arg, ...);
4  int execlp (const char *file, const char *arg, ...);
5  int execle (const char *path, const char *arg, ..., char *const envp[]);
6
7  int execv  (const char *path, char *const argv[]);
8  int execvp (const char *file, char *const argv[]);
9  int execve (const char *path, char *const argv[], char *const envp[]);
```

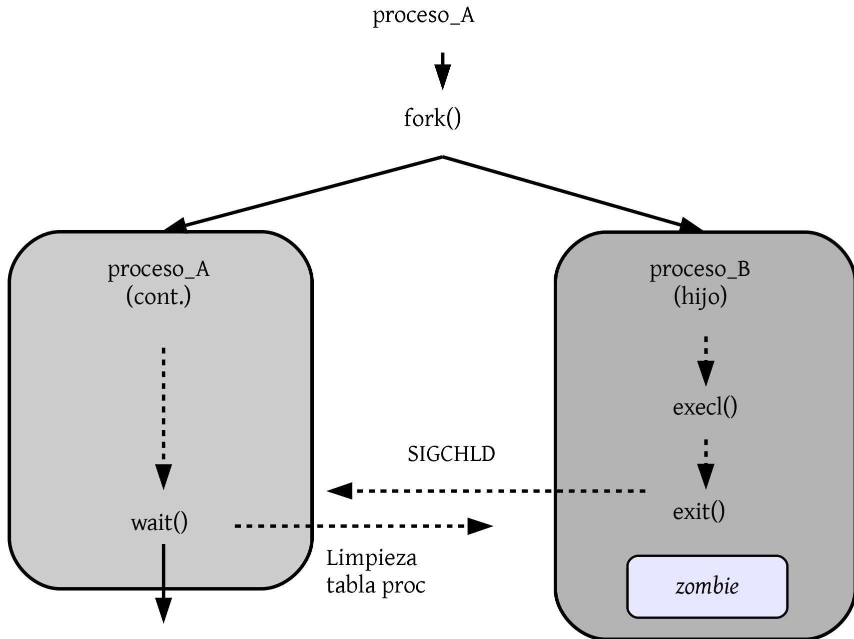


Figura 1.5: Esquema gráfico de la combinación *fork()* + *exec()*.

La llamada al sistema ***execl()*** tiene los siguientes parámetros:

1. La **ruta** al archivo que contiene el código binario a ejecutar por el proceso, es decir, el ejecutable asociado al nuevo segmento de código.

2. Una serie de **argumentos de línea de comandos**, terminado por un apuntador a NULL. El nombre del programa asociado al proceso que ejecuta *execl* suele ser el primer argumento de esta serie.

La llamada ***execlp*** tiene los mismos parámetros que *execl*, pero hace uso de la variable de entorno PATH para buscar el ejecutable del proceso. Por otra parte, *execle* es también similar a ***execl***, pero añade un parámetro adicional que representa el nuevo ambiente del programa a ejecutar. Finalmente, las llamadas ***execv*** difieren en la forma de pasar los argumentos de línea de comandos, ya que hacen uso de una sintaxis basada en arrays.

El listado 1.6 muestra la estructura de un programa encargado de la creación de una serie de procesos mediante la combinación de *fork()* y *execl()* dentro de una estructura de bucle.

Como se puede apreciar, el programa recoge por línea de órdenes el número de procesos que se crearán posteriormente[2]. Note cómo se hace uso de una estructura condicional para diferenciar entre el código asociado al proceso padre y al proceso hijo. Para ello, el valor devuelto por *fork()*, almacenado en la variable *childpid*, actúa de discriminante.

[2]Por simplificación no se lleva a cabo un control de errores

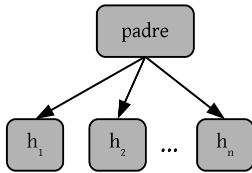


Figura 1.6: Representación gráfica de la creación de varios procesos hijo a partir de uno padre.

En el *case* 0 de la sentencia *switch* (línea 12) se ejecuta el código del hijo. Recuerde que *fork()* devuelve 0 al proceso hijo, por lo que en ese caso habrá que asociar el código del proceso hijo. En este ejemplo, dicho código reside en una nueva unidad ejecutable, la cual se encuentra en el directorio *exec* y se denomina *hijo*. Esta información es la que se usa cuando se llama a *execl()*. Note cómo este nuevo proceso acepta por línea de órdenes el número de procesos creados (*argv[1]*), recogido previamente por el proceso padre (línea 13).

Listado 1.6: Uso de fork+exec

```
1  #include <sys/types.h>
2  #include <unistd.h>
3  #include <stdlib.h>
4  #include <stdio.h>
5
6  int main (int argc, char *argv[]) {
7    pid_t childpid;
8    int n = atoi(argv[1]), i;
9
10   for (i = 1; i <= n; i++) {
11     switch(childpid = fork()) {
12     case 0:  // Código del hijo.
13       execl("./exec/hijo", "hijo", argv[1], NULL);
14       break; // Para evitar entrar en el for.
15     }
16   }
17
18   // Se obvia la espera a los procesos
19   // y la captura de eventos de teclado.
20
21   return 0;
22 }
```

El nuevo código del proceso hijo es trivial y simplemente imprime el ID del proceso y el número de *hermanos* que tiene. Este tipo de esquemas, en los que se estructura el código del proceso padre y de los hijos en ficheros fuente independientes, será el que se utilice en los temas sucesivos.

Compilación

La compilación de los programas asociados al proceso padre y al hijo, respectivamente, es independiente. En otras palabras, serán objetivos distintos que generarán ejecutables distintos.

En este punto, resulta interesante preguntarse sobre lo que sucede con el proceso padre después de la creación de los procesos hijo. Hasta ahora se ha planteado que ambos siguen su ejecución desde la instrucción siguiente a *fork()*. Sin embargo, si un padre desea esperar a que su hijo termine, entonces ha de ejecutar una llamada explícita a *wait()* o alguna función similar. De este modo, todo queda perfectamente controlado por parte del programador.

Listado 1.7: Código del proceso hijo

```
1  #include <sys/types.h>
2  #include <unistd.h>
3  #include <stdlib.h>
4  #include <stdio.h>
5
6  void hijo (const char *num_hermanos);
7
8  int main (int argc, char *argv[]) {
9    hijo(argv[1]);
10   return 0;
11 }
12
13 void hijo (const char *num_hermanos) {
14   printf("Soy %ld y tengo %d hermanos!\n", (long)getpid(), atoi(num_hermanos) - 1);
15 }
```

La llamada *wait()* detiene el proceso que llama hasta que un hijo de dicho proceso finalice o se detenga, o hasta que el proceso que realizó dicha llamada reciba otra señal (como la de terminación). Por ejemplo, el estándar POSIX define SIGCHLD como la señal enviada a un proceso cuando su proceso hijo finaliza su ejecución.

Otra señal esencial está representada por SIGINT, enviada a un proceso cuando el usuario desea interrumpirlo. Está señal se representa típicamente mediante la combinación Ctrl+C desde el terminal que controla el proceso.

Listado 1.8: Primitivas POSIX wait

```
1  #include <sys/types.h>
2  #include <sys/wait.h>
3
4  pid_t wait (int *status);
5  pid_t waitpid (pid_t pid, int *status, int options);
6
7  // BSD style
8  pid_t wait3 (int *status, int options, struct rusage *rusage);
9  pid_t wait4 (pid_t pid, int *status, int options, struct rusage *rusage);
```

El listado 1.9 muestra la inclusión del código necesario para esperar de manera adecuada la finalización de los procesos hijo y gestionar la terminación abrupta del proceso padre mediante la combinación de teclas Ctrl+C.

Listado 1.9: Uso de fork+exec+wait

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <wait.h>

#define NUM_HIJOS 5

void finalizarprocesos ();
void controlador (int senhal);

pid_t pids[NUM_HIJOS];

int main (int argc, char *argv[]) {
  int i;
  char num_hijos_str[10];

  if (signal(SIGINT, controlador) == SIG_ERR) {
    fprintf(stderr, "Abrupt termination.\n");
    exit(EXIT_FAILURE);
  }

  sprintf(num_hijos_str, "%d", NUM_HIJOS);
  for (i = 0; i < NUM_HIJOS; i++) {
    switch(pids[i] = fork()) {
    case 0:  // Código del hijo.
      execl("./exec/hijo", "hijo", num_hijos_str, NULL);
      break; // Para evitar entrar en el for.
    }
  }
  free(num_hijos_str);

  // Espera terminación de hijos...
  for (i = 0; i < NUM_HIJOS; i++) {
    waitpid(pids[i], 0, 0);
  }

  return EXIT_SUCCESS;
}

void finalizarprocesos () {
  int i;
  printf ("\n-------------- Finalizacion de procesos ------------- \n");
  for (i = 0; i < NUM_HIJOS; i++) {
    if (pids[i]) {
      printf ("Finalizando proceso [%d]... ", pids[i]);
      kill(pids[i], SIGINT); printf("<Ok>\n");
    }
  }
}

void controlador (int senhal) {
  printf("\nCtrl+c captured.\n"); printf("Terminating...\n\n");
  // Liberar recursos...
  finalizarprocesos(); exit(EXIT_SUCCESS);
}
```

En primer lugar, la espera a la terminación de los hijos se controla mediante las líneas [35-37] utilizando la primitiva *waitpid*. Esta primitiva se comporta de manera análoga a *wait*, aunque bloqueando al proceso que la ejecuta para esperar a otro proceso con un *pid* concreto. Note como previamente se ha utilizado un array auxiliar denominado *pids* (línea [13]) para almacenar el *pid* de todos y cada uno de los procesos hijos creados mediante *fork* (línea [26]). Así, sólo habrá que esperarlos después de haberlos lanzado.

Por otra parte, note cómo se captura la señal SIGINT, es decir, la terminación mediante *Ctrl+C*, mediante la función *signal* (línea [19]), la cual permite asociar la captura de una señal con una **función de retrollamada**. Ésta función definirá el código que se tendrá que ejecutar cuando se capture dicha señal. Básicamente, en esta última función, denominada *controlador*, se incluirá el código necesario para liberar los recursos previamente reservados, como por ejemplo la memoria dinámica, y para destruir los procesos hijo que no hayan finalizado.

Si *signal()* devuelve un código de error *SIG_ERR*, el programador es responsable de controlar dicha situación excepcional.

Listado 1.10: Primitiva POSIX signal

```
1 #include <signal.h>
2
3 typedef void (*sighandler_t)(int);
4
5 sighandler_t signal (int signum, sighandler_t handler);
```

1.1.3. Procesos e hilos

El modelo de proceso presentado hasta ahora se basa en un único flujo o hebra de ejecución. Sin embargo, los sistemas operativos modernos se basan en el principio de la **multiprogramación**, es decir, en la posibilidad de manejar distintas hebras o hilos de ejecución de manera simultánea con el objetivo de paralelizar el código e incrementar el rendimiento de la aplicación.

Sincronización

Conseguir que dos cosas ocurran al mismo tiempo se denomina comúnmente sincronización. En Informática, este concepto se asocia a las relaciones existentes entre eventos, como la serialización (A debe ocurrir antes que B) o la exclusión mutua (A y B no pueden ocurrir al mismo tiempo).

Esta idea también se plasma a nivel de lenguaje de programación. Algunos ejemplos representativos son los APIs de las bibliotecas de hilos *Pthread*, *Win32* o *Java*. Incluso existen bibliotecas de gestión de hilos que se enmarcan en capas software situadas sobre la capa del sistema operativo, con el objetivo de independizar el modelo de programación del propio sistema operativo subyacente.

En este contexto, una **hebra** o **hilo** se define como la unidad básica de utilización del procesador y está compuesto por los elementos:

- Un **ID de hebra**, similar al ID de proceso.
- Un **contador de programa**.

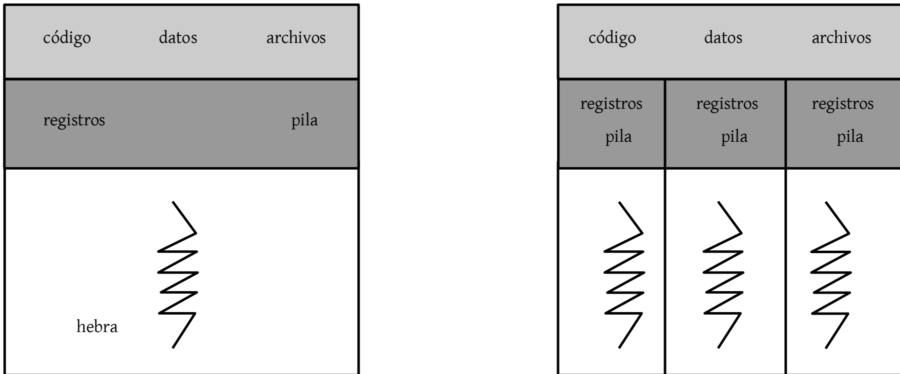


Figura 1.7: Esquema gráfico de los modelos monohilo y multihilo.

- Un conjunto de **registros**.
- Una **pila**.

Sin embargo, y **a diferencia de un proceso**, las hebras que pertenecen a un mismo proceso comparten la sección de código, la sección de datos y otros recursos proporcionados por el sistema operativo, como los manejadores de los archivos abiertos o las señales. Precisamente, esta diferencia con respecto a un proceso es lo que supone su principal ventaja.

Desde otro punto de vista, la idea de hilo surge de la posibilidad de compartir recursos y permitir el acceso concurrente a esos recursos. La unidad mínima de ejecución pasa a ser el hilo, asignable a un procesador. Los hilos tienen el mismo código de programa, pero cada uno recorre su propio camino con su PC, es decir, tiene su propia situación aunque dentro de un contexto de compartición de recursos.

Informalmente, un hilo se puede definir como un ***proceso ligero*** que tiene la misma funcionalidad que un *proceso pesado*, es decir, los mismos estados. Por ejemplo, si un hilo abre un fichero, éste estará disponible para el resto de hilos de una tarea.

En un procesador de textos, por ejemplo, es bastante común encontrar hilos para la gestión del teclado o la ejecución del corrector ortográficos, respetivamente. Otro ejemplo podría ser un servidor web que manejara hilos independientes para atender las distintas peticiones entrantes.

Las **ventajas de la programación multihilo** se pueden resumir en las tres siguientes:

- **Capacidad de respuesta**, ya que el uso de múltiples hilos proporciona un enfoque muy flexible. Así, es posible que un hilo se encuentra atendiendo una petición de E/S mientras otro continúa con la ejecución de otra funcionalidad distinta. Además, es posible plantear un esquema basado en el paralelismo no bloqueante en llamadas al sistema, es decir, un esquema basado en el bloqueo de un hilo a nivel individual.
- **Compartición de recursos**, posibilitando que varios hilos manejen el mismo espacio de direcciones.

- **Eficacia**, ya que tanto la creación, el cambio de contexto, la destrucción y la liberación de hilos es un orden de magnitud más rápida que en el caso de los procesos pesados. Recuerde que las operaciones más costosas implican el manejo de operaciones de E/S. Por otra parte, el uso de este tipo de programación en arquitecturas de varios procesadores (o núcleos) incrementa enormemente el rendimiento de la aplicación.

Caso de estudio. POSIX *Pthreads*

Pthreads es un estándar POSIX que define un **interfaz** para la creación y sincronización de hilos. Recuerde que se trata de una especificación y no de una implementación, siendo ésta última responsabilidad del sistema operativo.

El mánejo básico de hilos mediante *Pthreads* implica el uso de las primitivas de creación y de espera que se muestran en el listado 1.11. Como se puede apreciar, la llamada *pthread_create()* necesita una función que defina el código de ejecución asociado al propio hilo (definido en el tercer parámetro). Por otra parte, *pthread_join()* tiene un propósito similar al ya estudiado en el caso de la llamada *wait()*, es decir, se utiliza para que la *hebra padre* espera a que la *hebra hijo* finalice su ejecución.

POSIX

Portable Operating System Interface es una familia de estándares definidos por el comité IEEE con el objetivo de mantener la portabilidad entre distintos sistemas operativos. La *X* de *POSIX* es una referencia a sistemas *Unix*.

Listado 1.11: Primitivas POSIX Pthreads

```
1 #include <pthread.h>
2
3 int pthread_create (pthread_t *thread,
4                     const pthread_attr_t *attr,
5                     void *(*start_routine) (void *),
6                     void *arg);
7
8 int pthread_join   (pthread_t thread, void **retval);
```

El listado de código 1.12 muestra un ejemplo muy sencillo de uso de *Pthreads* en el que se crea un hilo adicional que tiene como objetivo realizar el sumatorio de todos los números inferiores o iguales a uno pasado como parámetro. La función *mi_hilo()* (líneas 27-37) es la que realmente implementa la funcionalidad del hilo creado mediante *pthread_create()* (línea 19). El resultado se almacena en una variable definida globalmente. Para llevar a cabo la compilación y ejecución de este ejemplo, es necesario enlazar con la biblioteca *pthreads*:

```
$ gcc -lpthread thread_simple.c -o thread_simple
$ ./thread_simple <valor>
```

El resultado de la ejecución de este programa para un valor, dado por línea de órdenes, de 7 será el siguiente:

```
$ Suma total: 28.
```

Listado 1.12: Ejemplo de uso de Pthreads

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

int suma;
void *mi_hilo (void *valor);

int main (int argc, char *argv[]) {
  pthread_t tid;
  pthread_attr_t attr;

  if (argc != 2) {
    fprintf(stderr, "Uso: ./pthread <entero>\n");
    return -1;
  }

  pthread_attr_init(&attr); // Att predeterminados.
  // Crear el nuevo hilo.
  pthread_create(&tid, &attr, mi_hilo, argv[1]);
  pthread_join(tid, NULL);  // Esperar finalización.

  printf("Suma total: %d.\n", suma);

  return 0;
}

void *mi_hilo (void *valor) {
  int i, ls;
  ls = atoi(valor);
  i = 0, suma = 0;

  while (i <= ls) {
    suma += (i++);
  }

  pthread_exit(0);
}
```

1.2. Fundamentos de programación concurrente

Un **proceso cooperativo** es un proceso que puede verse afectado por otros procesos que se estén ejecutando. Estos procesos pueden compartir únicamente datos, intercambiados mediante algún mecanismo de envío de mensajes o mediante el uso de archivos, o pueden compartir datos y código, como ocurre con los hilos o procesos ligeros.

En cualquier caso, el acceso concurrente a datos compartidos puede generar **inconsistencias** si no se usa algún tipo de esquema que garantice la coherencia de los mismos. A continuación, se discute esta problemática y se plantean brevemente algunas soluciones, las cuales se estudiarán con más detalle en sucesivos capítulos.

1.2.1. El problema del productor/consumidor

Considere un buffer de tamaño limitado que se comparte por un proceso productor y otro proceso consumidor. Mientras el primero añade elementos al espacio de almacenamiento compartido, el segundo los consume. Evidentemente, el acceso concurrente a este buffer puede generar incosistencias de dichos datos a la hora de, por ejemplo, modificar los elementos contenidos en el mismo.

Suponga que también se mantiene una variable *cont* que se incrementa cada vez que se añade un elemento al buffer y se decrementa cuando se elimina un elemento del mismo. Este esquema posibilita manejar *buffer_size* elementos, en lugar de uno menos si dicha problemática se controla con dos variables *in* y *out*.

Multiprocesamiento

Recuerde que en los SSOO actuales, un único procesador es capaz de ejecutar múltiples hilos de manera simultánea. Si el computador es paralelo, entonces existirán múltiples núcleos físicos ejecutando código en paralelo.

Aunque el código del productor y del consumidor es correcto de manera independiente, puede que no funcione correctamente al ejecutarse de manera simultánea. Por ejemplo, la **ejecución concurrente** de las operaciones de incremento y decremento del contador pueden generar inconsistencias, tal y como se aprecia en la figura 1.8, en la cual se desglosan dichas operaciones en instrucciones máquina. En dicha figura, el código de bajo nivel asociado a incrementar y decrementar el contador inicializado a 5, genera un resultado inconsistente, ya que esta variable tendría un valor final de 4, en lugar de 5.

Este estado incorrecto se alcanza debido a que la manipulación de la variable contador se realiza de manera concurrente por dos procesos independientes sin que exista ningún mecanismos de sincronización. Este tipo de situación, en la que se produce un acceso concurrente a unos datos compartidos y el resultado de la ejecución depende del orden de las instrucciones, se denomina **condición de carrera**. Para evitar este tipo de situaciones se ha de garantizar que sólo un proceso pueda manipular los datos compartidos de manera simultánea, es decir, dichos procesos se han de sincronizar de algún modo.

Este tipo de situaciones se producen constantemente en el ámbito de los sistemas operativos, por lo que el uso de mecanismos de sincronización es crítico para evitar problemas potenciales.

1.2.2. La sección crítica

El segmento de código en el que un proceso puede modificar variables compartidas con otros procesos se denomina **sección crítica** (ver figura 1.9). Para evitar inconsistencias, una de las ideas que se plantean es que cuando un proceso está ejecutando su sección crítica ningún otro procesos puede ejecutar su sección crítica asociada.

El **problema de la sección crítica** consiste en diseñar algún tipo de solución para garantizar que los procesos involucrados puedan operar sin generar ningún tipo de inconsistencia. Una posible estructura para abordar esta problemática se plantea en la figura 1.9, en la que el código se divide en las siguientes secciones:

- **Sección de entrada**, en la que se solicita el acceso a la sección crítica.

```
while (1) {
  // Produce en nextP.
  while (cont == N);
       // No hacer nada.
  buffer[in] = nextP;
  in = (in + 1) % N;
  cont++;
}
```

```
while (1) {
  while (cont == 0);
       // No hacer nada.
  nextC = buffer[out];
  out = (out + 1) % N;
  cont--;
  // Consume nextC.
}
```

```
cont++
-----------
r1 = cont
r1 = r1 + 1
cont = r1

cont--
-----------
r2 = cont
r2 = r2 - 1
cont = r2
```

```
p: r1 = cont
p: r1 = r1 + 1
c: r2 = cont
c: r2 = r2 - 1
p: cont = r1
c: cont = r2
```

Figura 1.8: Código para los procesos productor y consumidor.

- **Sección crítica**, en la que se realiza la modificación efectiva de los datos compartidos.
- **Sección de salida**, en la que típicamente se hará explícita la salida de la sección crítica.
- **Sección restante**, que comprende el resto del código fuente.

Figura 1.9: Estructura general de un proceso.

Normalmente, la sección de entrada servirá para manipular algún tipo de mecanismo de sincronización que garantice el acceso exclusivo a la sección crítica de un proceso. Mientras tanto, la sección de salida servirá para notificar, mediante el mecanismo de sincronización correspondiente, la salida de la sección crítica. Este hecho, a su vez, permitirá que otro proceso pueda acceder a su sección crítica.

Cualquier solución al problema de la sección crítica ha de satisfacer los siguientes requisitos:

1. **Exclusión mutua**, de manera que si un proceso p_i está en su sección crítica, entonces ningún otro proceso puede ejecutar su sección crítica.
2. **Progreso**, de manera que sólo los procesos que no estén en su sección de salida, suponiendo que ningún proceso ejecutase su sección crítica, podrán participar en la decisión de quién es el siguiente en ejecutar su sección crítica. Además, la toma de esta decisión ha de realizarse en un tiempo limitado.
3. **Espera limitada**, de manera que existe un límite en el número de veces que se permite entrar a otros procesos a sus secciones críticas antes de que otro proceso haya solicitado entrar en su propia sección crítica (y antes de que le haya sido concedida).

Las soluciones propuestas deberían ser independientes del número de procesos, del orden en el que se ejecutan las instrucciones máquinas y de la velocidad relativa de ejecución de los procesos.

Una posible solución para N procesos sería la que se muestra en el listado de código 1.13. Sin embargo, esta solución no sería válida ya que no cumple el principio de **exclusión mutua**, debido a que un proceso podría entrar en la sección crítica si se produce un cambio de contexto justo después de la sentencia *while* (línea 5).

Listado 1.13: Solución deficiente para N procesos

```
1  int cerrojo = 0;
2
3  do {
4    // SECCIÓN DE ENTRADA
5    while (cerrojo == 1); // No hacer nada.
6    cerrojo = 1;
7    // SECCIÓN CRÍTICA.
8    // ...
9    // SECCIÓN DE SALIDA.
10   cerrojo = 0;
11   // SECCIÓN RESTANTE.
12  }while (1);
```

Otro posible planteamiento para **dos procesos** se podría basar en un esquema de alternancia entre los mismos, modelado mediante una variable booleana *turn*, la cual indica qué proceso puede entrar en la sección crítica, es decir, si *turn* es igual a i, entonces el proceso p_i podrá entrar en su sección crítica ($j == (i - 1)$).

Listado 1.14: Solución 2 procesos mediante alternancia

```
1  do {
2    // SECCIÓN DE ENTRADA
3    while (turn != i); // No hacer nada.
4    // SECCIÓN CRÍTICA.
5    // ...
6    // SECCIÓN DE SALIDA.
7    turn = j;
8    // SECCIÓN RESTANTE.
9   }while (1);
```

El problema de esta solución es que cuando p_j abandona la sección crítica y ha modificado *turn* a i, entonces no puede volver a entrar en ella porque primero tiene que entrar p_i. Éste proceso podría seguir ejecutando su sección restante, por lo que la **condición de progreso** no se cumple. Sin embargo, la exclusión mutua sí que se satisface.

Otra posible solución para la problemática de dos procesos podría consistir en hacer uso de un array de booleanos, originalmente inicializados a *false*, de manera que si $flag[i]$ es igual a *true*, entonces el proceso p_i está preparado para acceder a su sección crítica.

Aunque este planteamiento garantiza la exclusión mutua, no se satisface la **condición de progreso**. Por ejemplo, si p_0 establece su turno a *true* en la sección de entrada y, a continuación, hay un cambio de contexto, el proceso p_1 puede poner a *true* su turno de manera que ambos se queden bloqueados sin poder entrar en la sección crítica. En otras palabras, la decisión de quién entra en la sección crítica se pospone indefinidamente, violando la condición de progreso e impidiendo la evolución de los procesos.

Listado 1.15: Solución 2 procesos con array de booleanos

```
1  do {
2    // SECCIÓN DE ENTRADA
3    flag[i] = true;
4    while (flag[j]); // No hacer nada.
5    // SECCIÓN CRÍTICA.
6    // ...
7    // SECCIÓN DE SALIDA.
8    flag[i] = false;
9    // SECCIÓN RESTANTE.
10  }while (1);
```

¿Qué ocurre si se invierte el orden de las operaciones de la SE?

Listado 1.16: Solución 2 procesos con array de booleanos

```
1  do {
2    // SECCIÓN DE ENTRADA
3    while (flag[j]); // No hacer nada.
4    flag[i] = true;
5    // SECCIÓN CRÍTICA.
6    // SECCIÓN DE SALIDA.
7    flag[i] = false;
8    // SECCIÓN RESTANTE.
9   }while (1);
```

Esta solución no cumpliría con el principio de **exclusión mutua**, ya que los dos procesos podrían ejecutar su sección crítica de manera concurrente.

Solución de Peterson (2 procesos)

En este apartado se presenta una solución al problema de la sincronización de dos procesos que se basa en el algoritmo de Peterson, en honor a su inventor, y que fue planteado en 1981. La solución de Peterson se aplica a dos procesos que van alternando la ejecución de sus respectivas secciones críticas y restantes. Dicha solución es una mezcla de las dos soluciones propuestas anteriormente.

Los dos procesos comparten tanto el array *flag*, que determina los procesos que están listos para acceder a la sección crítica, como la variable *turn*, que sirve para determinar el proceso que accederá a su sección crítica. Esta **solución es correcta** para dos procesos, ya que satisface las tres condiciones anteriormente mencionadas.

Para entrar en la sección crítica, el proceso p_i asigna *true* a *flag[i]* y luego asigna a *turn* el valor j, de manera que si el proceso p_j desea entrar en su sección crítica, entonces puede hacerlo. Si los dos procesos intentan acceder al mismo tiempo, a la variable *turn* se le asignarán los valores i y j (o viceversa) en un espacio de tiempo muy corto, pero sólo prevalecerá una de las asignaciones (la otra se sobreescribirá). Este valor determinará qué proceso accederá a la sección crítica.

Listado 1.17: Solución 2 procesos; algoritmo de Peterson

```
1  do {
2    // SECCIÓN DE ENTRADA
3    flag[i] = true;
4    turn = j;
5    while (flag[j] && turn == j); // No hacer nada.
6    // SECCIÓN CRÍTICA.
7    // ...
8    // SECCIÓN DE SALIDA.
9    flag[i] = false;
10   // SECCIÓN RESTANTE.
11  }while (1);
```

A continuación se demuestra que si p_1 está en su sección crítica, entonces p_2 no está en la suya.

```
1. p1 en SC                                           (premisa)
2. p1 en SC --> flag[1]=true y (flag[2]=false o turn!=2) (premisa)
3. flag[1]=true y (flag[2]=false o turn!=2)           (MP 1,2)
4. flag[2]=false o turn!=2  (A o B)                   (EC 3)

Demostrando A
5. flag[2]=false             (premisa)
6. flag[2]=false --> p2 en SC (premisa)
7. p2 en SR (p2 no SC)       (MP 5,6)

Demostrando B
8.  turn=!2                               (premisa)
9.  flag[1]=true                          (EC 3)
10. flag[1]=true y turn=1                 (IC 8,9)
11. (flag[1]=true y turn=1) --> p2 en SE (premisa)
12. p2 en SE (p2 no SC)                   (MP 10,11)
```

Solución de Lamport (n procesos)

El algoritmo para n procesos desarrollado por Lamport es una solución general e independiente del número de procesos inicialmente concebida para entornos distribuidos. Los procesos comparten dos arrays, uno de booleanos denominado *eleccion*, con sus elementos inicializados a *false*, y otro de enteros denominado *num*, con sus elementos inicializados a 0.

Este planteamiento se basa en que si p_i está en la sección crítica y p_j intenta entrar, entonces p_j ejecuta el segundo bucle *while* y detecta que $num[i] \neq 0$, garantizando así la exclusión mutua.

Listado 1.18: Solución n procesos; algoritmo de Lamport

```
1  do {
2    // SECCIÓN DE ENTRADA
3    eleccion[i] = true;
4    num[i] = max(num[0], ..., num[n]) + 1;
5    eleccion[i] = false;
6    for (j = 0; j < n; j++) {
7      while (eleccion[j]);
8      while (num[j] != 0 && (num[j], j) < (num[i], i));
9    }
10   // SECCIÓN CRÍTICA.
11   // ...
12   // SECCIÓN DE SALIDA.
13   num[i] = 0;
14   // SECCIÓN RESTANTE.
15  }while (1);
```

Soluciones hardware

Figura 1.10: Acceso a la sección crítica mediante un cerrojo.

En general, cualquier solución al problema de la sección crítica se puede abstraer en el uso de un mecanismo simple basado en **cerrojos**. De este modo, las condiciones de carrera no se producirían, siempre y cuando antes de acceder a la sección crítica, un proceso adquiriese el uso exclusivo del cerrojo, bloqueando al resto de procesos que intenten abrirlo. Al terminar de ejecutar la sección crítica, dicho cerrojo quedaría liberado en la sección de salida. Este planteamiento se muestra de manera gráfica en la figura 1.10.

Evidentemente, tanto las operaciones de adquisición y liberación del cerrojo han de ser **atómicas**, es decir, su ejecución ha de producirse en una unidad de trabajo indivisible.

El problema de la sección crítica se podría solucionar fácilmente deshabilitando las interrupciones en entornos con un único procesador. De este modo, sería posible asegurar un esquema basado en la ejecución en orden de la secuencia actual de instrucciones, sin permitir la posibilidad de desalojo, tal y como se plantea en los *kernels* no apropiativos. Sin embargo, esta solución no es factible en entornos multiprocesador, los cuales representan el caso habitual en la actualidad.

Como ejemplo representativo de soluciones hardware, una posibilidad consiste en hacer uso de una instrucción **swap**, ejecutada de manera atómica mediante el hardware correspondiente, como parte de la sección de entrada para garantizar la condición de exclusión mutua.

Listado 1.19: Uso de operaciones swap

```
1  do {
2    // SECCIÓN DE ENTRADA
3    key = true;
4    while (key == true)
5      swap(&lock, &key);
6    // SECCIÓN CRÍTICA.
7    // ...
8    // SECCIÓN DE SALIDA.
9    lock = false;
10   // SECCIÓN RESTANTE.
11  }while (1);
```

Consideraciones

Las soluciones estudiadas hasta ahora no son, por lo general, legibles y presentan dificultadas a la hora de extenderlas a problemas más generales en los que se manejen n procesos.

Por otra parte, los procesos que intentan acceder a la sección crítica se encuentra en un estado de **espera activa**, normalmente modelado mediante bucles que comprueban el valor de una variable de manera ininterrumpida. Este planteamiento es muy ineficiente y no se puede acoplar en sistemas multiprogramados.

La consecuencia directa de estas limitaciones es la necesidad de plantear mecanismos que sean más sencillos y que sean independientes del número de procesos que intervienen en un determinado problema.

1.2.3. Mecanismos básicos de sincronización

En este apartado se plantean tres mecanismos de sincronización que se estudiarán con detalle en los capítulos 2, 3 y 4, respectivamente. Estos mecanismos son los semáforos, el paso de mensajes y los monitores.

Semáforos

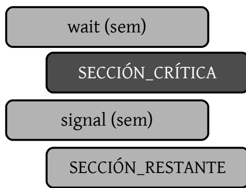


Figura 1.11: Acceso a la sección crítica mediante un semáforo.

Un semáforo es un mecanismo de sincronización que encapsula una variable entera que sólo es accesible mediante dos operaciones atómicas estándar: **wait** y **signal**. La inicialización de este tipo abstracto de datos también se considera relevante para definir el comportamiento inicial de una solución. Recuerde que la modificación del valor entero encapsulado dentro de un semáforo ha de modificarse mediante instrucciones atómicas, es decir, tanto *wait* como *signal* han de ser instrucciones atómicas. Típicamente, estas operaciones se utilizarán para gestionar el acceso a la sección crítica por un número indeterminado de procesos.

La definición de *wait* es la siguiente:

Listado 1.20: Semáforos; definición de wait

```
1 wait (sem) {
2   while (sem <= 0); // No hacer nada.
3   sem--;
4 }
```

Por otra parte, la definición de *signal* es la siguiente:

Listado 1.21: Semáforos; definición de signal

```
1 signal (sem) {
2   sem++;
3 }
```

A continuación se enumeran algunas **características** de los semáforos y las implicaciones que tiene su uso en el contexto de la sincronización entre procesos.

- Sólo es posible acceder a la variable del semáforo mediante *wait* o *signal*. No se debe asignar o comparar los valores de la variable encapsulada en el mismo.
- Los semáforos han de inicializarse con un valor no negativo.
- La operación *wait* decrementa el valor del semáforo. Si éste se hace negativo, entonces el proceso que ejecuta *wait* se bloquea.
- La operación *signal* incrementa el valor del semáforo. Si el valor no era positivo, entonces se desbloquea a un proceso bloqueado previamente por *wait*.

Estas características tienen algunas implicaciones importantes. Por ejemplo, no es posible determinar a priori si un proceso que ejecuta *wait* se quedará bloqueado o no tras su ejecución. Así mismo, no es posible conocer si después de *signal* se despertará algún proceso o no.

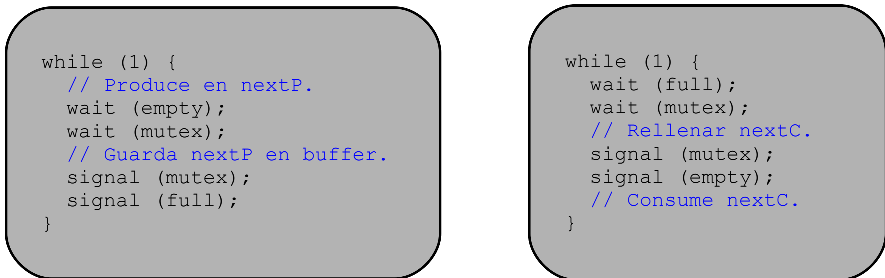


Figura 1.12: Procesos productor y consumidor sincronizados mediante un semáforo.

Por otra parte, *wait* y *signal* son mutuamente excluyentes, es decir, si se ejecutan de manera concurrente, entonces se ejecutarán secuencialmente y en un orden no conocido a priori.

Si el valor de un semáforo es positivo, dicho valor representa el número de procesos que pueden decrementarlo a través de *wait* sin quedarse bloqueados. Por el contrario, si su valor es negativo, representa el número de procesos que están bloqueados. Finalmente, un valor igual a cero implica que no hay procesos esperando, pero una operación *wait* hará que un proceso se bloquee.

Los semáforos se suelen clasificar en contadores y binarios, en función del rango de valores que tome las variables que encapsulan. Un **semáforo binario** es aquél que sólo toma los valores 0 ó 1 y también se conoce como cerrojo *mutex* (*mutual exclusion*). Un **semáforo contador** no cumple esta restricción.

Las principales ventajas de los semáforos con respecto a otro tipo de mecanismos de sincronización se pueden resumir en las tres siguientes: i) **simplicidad**, ya que facilitan el desarrollo de soluciones de concurrencia y son simples, ii) **correctitud**, ya que los soluciones son generalmente limpias y legibles, siendo posible llevar a cabo demostraciones formales, iii) **portabilidad**, ya que los semáforos son altamente portables en diversas plataformas y sistemas operativos.

La figura 1.12 muestra una posible solución al problema del productor/consumidor. Típicamente los semáforos se utilizan tanto para gestionar el sincronismo entre procesos como para controlar el acceso a fragmentos de memoria compartida, como por ejemplo el buffer de productos.

Esta solución se basa en el uso de tres semáforos:

- **mutex** se utiliza para proporcionar exclusión mutua para el acceso al buffer de productos. Este semáforo se inicializa a 1.
- **empty** se utiliza para controlar el número de huecos vacíos del buffer. Este semáforo se inicializa a n.
- **full** se utiliza para controlar el número de huecos llenos del buffer. Este semáforo se inicializa a 0.

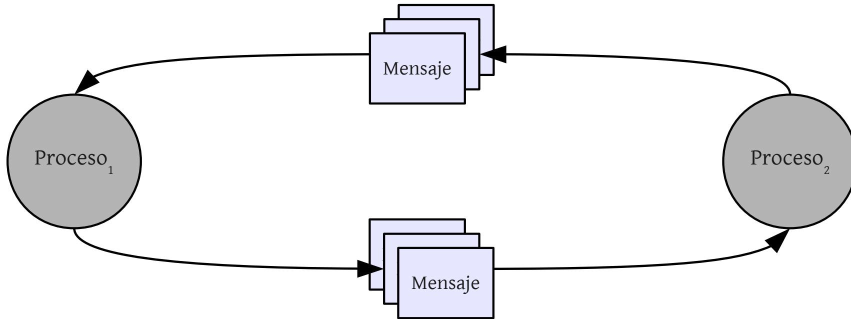


Figura 1.13: Esquema gráfico del mecanismo de paso de mensajes.

En el capítulo 2 se discutirá más en detalle el concepto de semáforo y su utilización en problemas clásicos de sincronización. Además, se estudiará una posible implementación utilizando las primitivas proporcionadas por el estándar POSIX.

Paso de mensajes

El mecanismo de paso de mensajes es otro esquema que permite la sincronización entre procesos. La principal diferencia con respecto a los semáforos es que permiten que los procesos se comuniquen sin tener que recurrir al uso de varibles de memoria compartida.

En los sistemas de memoria compartida hay poca cooperación del sistema operativo, están orientados al proceso y pensados para entornos centralizados, y es tarea del desarrollador establecer los mecanismos de comunicación y sincronización. Sin embargo, los sistemas basados en el paso de mensajes son válidos para entornos distribuidos, es decir, para sistemas con espacios lógicos distintos, y facilitar los mecanismos de comunicación y sincronización es labor del sistema operativo, sin necesidad de usar memoria compartida.

El paso de mensajes se basa en el uso de los dos primitivas siguientes:

- **send**, que permite el envío de información a un proceso.
- **receive**, que permite la recepción de información por parte de un proceso.

Además, es necesario un canal de comunicación entre los propios procesos para garantizar la entrega y recepción de los mensajes.

Resulta importante distinguir entre varios **tipos de comunicación**, en base a los conceptos directa/indirecta y simétrica/asimétrica. En esencia, la comunicación directa implica que en el mensaje se conocen, implícitamente, el receptor y el receptor del mismo. Por el contrario, la comunicación indirecta se basa en el uso de buzones. La comunicación simétrica se basa en que el receptor conoce de quién recibe un mensaje. Por el contrario, en la comunicación asímetrica el receptor puede recibir de cualquier proceso.

El listado de código 1.22 muestra un ejemplo básico de sincronización entre dos procesos mediante el paso de mensajes. En el capítulo 3 se discutirá más en detalle el concepto de paso de mensajes y su utilización en problemas clásicos de sincronización. Además, se estudiará una posible implementación utilizando las primitivas que el estándar POSIX proporciona.

Listado 1.22: Sincronización mediante paso de mensajes

```
// Proceso 1
while (1) {
 // Trabajo previo p1...
 send ('*', P2)
 // Trabajo restante p1...
}

// Proceso 2
while (1) {
 // Trabajo previo p2...
 receive (&msg, P1)
 // Trabajo restante p2...
}
```

Monitores

Aunque los semáforos tienen ciertas ventajas que garantizan su éxito en los problemas de sincronización entre procesos, también sufren ciertas debilidades. Por ejemplo, es posible que se produzcan errores de temporización cuando se usan. Sin embargo, la debilidad más importante reside en su propio uso, es decir, es fácil que un programador cometa algún error al, por ejemplo, intercambiar erróneamente un *wait* y un *signal*.

Con el objetivo de mejorar los mecanismos de sincronización y evitar este tipo de problemática, en los últimos años se han propuesto **soluciones de más alto nivel**, como es el caso de los monitores.

Básicamente, un tipo *monitor* permite que el programador defina una serie de operaciones públicas sobre un tipo abstracto de datos que gocen de la característica de la exclusión mutua. La idea principal está ligada al concepto de encapsulación, de manera que un procedimiento definido dentro de un monitor sólo puede acceder a las variables que se declaran como privadas o locales dentro del monitor.

Esta estructura garantiza que sólo un proceso esté activo cada vez dentro del monitor. La consecuencia directa de este esquema es que el programador no tiene que implementar de manera explícita esta restricción de sincronización. Esta idea se traslada también al concepto de **variables de condición**.

En esencia, un monitor es un mecanismo de sincronización de más alto nivel que, al igual que un cerrojo, protege la sección crítica y garantiza que solamente pueda existir un hilo activo dentro de la misma. Sin embargo, un monitor permite suspender un hilo dentro de la sección crítica posibilitando que otro hilo pueda acceder a la misma. Este segundo

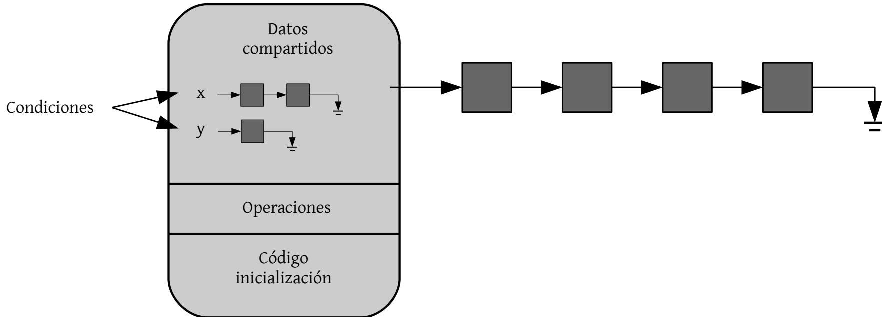


Figura 1.14: Esquema gráfico del uso de un monitor.

hilo puede abandonar el monitor, liberándolo, o suspenderse dentro del monitor. De cualquier modo, el hilo original se despierta y continua su ejecución dentro del monitor. Este esquema es escalable a múltiples hilos, es decir, varios hilos pueden suspenderse dentro de un monitor.

Los monitores proporcionan un mecanismo de sincronización **más flexible** que los cerrojos, ya que es posible que un hilo compruebe una condición y, si ésta es falsa, el hijo se pause. Si otro hilo cambia dicha condición, entonces el hilo original continúa su ejecución.

El siguiente listado de código muestra el uso del mecanismo de tipo monitor que el lenguaje Java proporciona para la sincronización de hebras. En Java, cualquier objeto tiene asociado un cerrojo. Cuando un método se declara como *synchronized*, la llamada al método implica la adquisición del cerrojo, evitando el acceso concurrente a cualquier otro método de la clase que use dicha declaración.

En el ejemplo que se expone en el listado 1.23, el monitor se utiliza para garantizar que el acceso y la modificación de la variable de clase *_edad* sólo se haga por otro hilo garantizando la exclusión mutua. De este modo, se garantiza que no se producirán inconsistencias al manipular el estado de la clase *Persona*.

Listado 1.23: Sincronización con monitores en Java

```
1  class Persona {
2      private int _edad;
3      public Persona (int edad) {
4      _edad = edad;
5      }
6      public synchronized void setEdad (int edad) {
7      _edad = edad;
8      }
9      public synchronized int getEdad () {
10     return _edad;
11     }
12 }
```

1.2.4. Interbloqueos y tratamiento del *deadlock*

En un entorno multiprogramado, los procesos pueden competir por un número limitado de recursos. Cuando un proceso intenta adquirir un recurso que está siendo utilizado por otro proceso, el primero pasa a un estado de espera, hasta que el recurso quede libre. Sin embargo, puede darse la situación en la que un proceso está esperando por un recurso que tiene otro proceso, que a su vez está esperando por un tercer recurso. A este tipo de situaciones se les denomina **interbloqueos** o ***deadlocks***.

La figura 1.15 muestra un ejemplo de interbloqueo en el que el proceso p_1 mantiene ocupado el recurso r_1 y está a la espera del recurso r_2, el cual está ocupado por el proceso p_2. Al mismo tiempo, p_2 está a la espera del recurso r_1. Debido a que los dos procesos necesitan los dos recursos para completar sus tareas, ambos se encuentran en una situación de interbloqueo a la espera del recurso necesario.

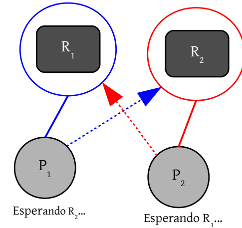


Figura 1.15: Situación de *deadlock* generada por dos procesos.

El modelo de sistema con el que se trabajará a continuación es una abstracción de la problemática de los sistemas operativos modernos, los cuales están compuestos de una serie de recursos por los que compiten una serie de procesos. En esencia, los recursos se clasifican en varios tipos y cada uno de ellos tiene asociado una serie de instancias. Por ejemplo, si el procesador tiene dos núcleos de procesamiento, entonces el recurso procesador tiene dos instancias asociadas.

Un conjunto de procesos estará en un estado de interbloqueo cuando todos los procesos del conjunto estén esperando a que se produzca un suceso que sólo puede producirse como resultado de la actividad de otro proceso del conjunto [12].

Típicamente, un proceso ha de solicitar un recurso antes de utilizarlo y ha de liberarlo después de haberlo utilizado. En modo de operación normal, un proceso puede utilizar un recurso siguiendo la siguiente secuencia:

1. **Solicitud**: si el proceso no puede obtenerla inmediatamente, tendrá que esperar hasta que pueda adquirir el recurso.
2. **Uso**: el proceso ya puede operar sobre el recurso.
3. **Liberación**: el proceso libera el recurso.

La solicitud y liberación de recursos se traducen en llamadas al sistema. Un ejemplo representativo sería la dupla *allocate()-free()*.

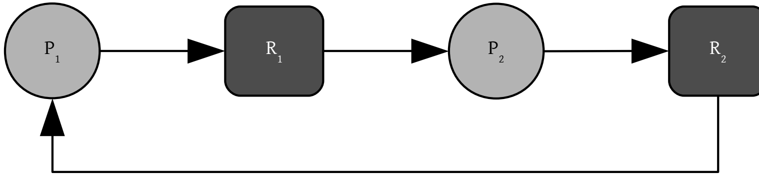


Figura 1.16: Condición de espera circular con dos procesos y dos recursos.

Los interbloqueos pueden surgir si se dan, de manera simultánea, las **cuatro condiciones de Coffman** [12]:

1. **Exclusión mutua**: al menos un recurso ha de estar en modo *no compartido*, es decir, sólo un proceso puede usarlo cada vez. En otras palabras, si el recurso está siendo utilizado, entonces el proceso que desee adquirirlo tendrá que esperar.
2. **Retención y espera**: un proceso ha de estar reteniendo al menos un recurso y esperando para adquirir otros recursos adicionales, actualmente retenidos por otros procesos.
3. **No apropiación**: los recursos no pueden desalojarse, es decir, un recurso sólo puede ser liberado, de manera voluntaria, por el proceso que lo retiene después de finalizar la tarea que estuviera realizando.
4. **Espera circular**: ha de existir un conjunto de procesos en competición $P = \{p_0, p_1, ..., p_{n-1}\}$ de manera que p_0 espera a un recurso retenido por p_1, p_1 a un recurso retenido por p_2, ..., y p_{n-1} a un recurso retenido por p_0.

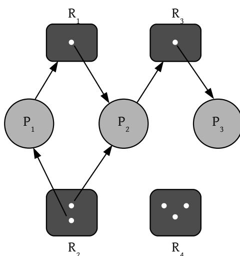


Figura 1.17: Ejemplo de grafo de asignación de recursos.

Todas estas condiciones han de darse por separado para que se produzca un interbloqueo, aunque resulta importante resaltar que existen dependencias entre ellas. Por ejemplo, la condición de *espera circular* implica que se cumpla la condición de *retención y espera*.

Grafos de asignación de recursos

Los interbloqueos se pueden definir de una forma más precisa, facilitando su análisis, mediante los grafos de asignación de recursos. El conjunto de nodos del grafo se suele dividir en dos subconjuntos, uno para los procesos activos del sistema $P = \{p_0, p_1, ..., p_{n-1}\}$ y otro formado por los tipos de recursos $R = \{r_1, r_2, ..., r_{n-1}\}$.

Por otra parte, el conjunto de aristas del grafo dirigido permite establecer las relaciones existentes entre los procesos y los recursos. Así, una arista dirigida del proceso p_i al recurso r_j significa que el proceso p_i ha solicitado una instancia del recurso r_j. Recuerde que en el modelo de sistema planteado cada recurso tiene asociado un número de instancias. Esta relación es una **arista de solicitud** y se representa como $p_i \rightarrow r_j$.

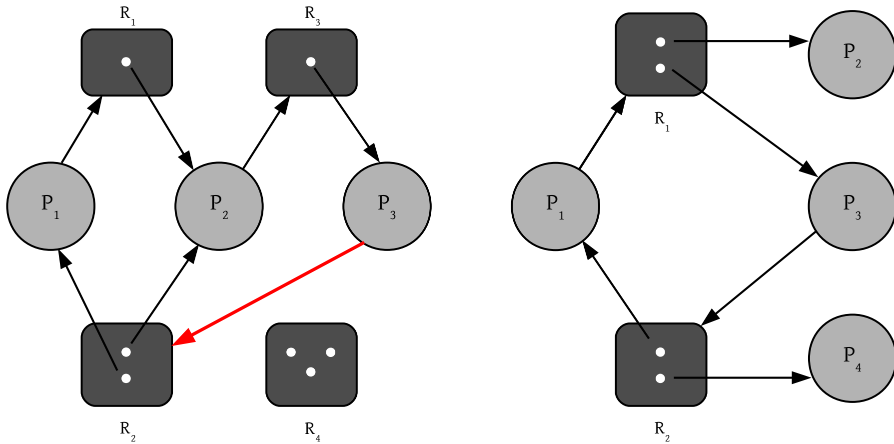


Figura 1.18: Grafos de asignación de recursos con *deadlock* (izquierda) y sin *deadlock* (derecha).

Por el contrario, una arista dirigida del recurso r_j al proceso p_i significa que se ha asignado una instancia del recurso r_j al proceso p_i. Esta **arista de asignación** se representa como $r_j \rightarrow p_i$. La figura 1.17 muestra un ejemplo de grafo de asignación de recursos.

Los grafos de asignación de recursos que no contienen ciclos no sufren interbloqueos. Sin embargo, la presencia de un ciclo puede generar la existencia de un interbloqueo, aunque no necesariamente. Por ejemplo, en la parte izquierda de la figura 1.18 se muestra un ejemplo de grafo que contiene un ciclo que genera una situación de interbloqueo, ya que no es posible romper el ciclo. Note cómo se cumplen las cuatro condiciones de *Coffman*.

La parte derecha de la figura 1.18 muestra la existencia de un ciclo en un grafo de asignación de recursos que no conduce a un interbloqueo, ya que si los procesos p_2 o p_4 finalizan su ejecución, entonces el ciclo se rompe.

Tratamiento del *deadlock*

Desde un punto de vista general, existen tres formas para llevar a cabo el tratamiento de los interbloqueos o *deadlocks*:

1. **Impedir o evitar** los interbloqueos, sin que se produzcan.
2. **Detectar y recuperarse** de los interbloqueos, es decir, tener mecanismos para identificarlos y, posteriormente, solventarlos.
3. **Ignorar** los interbloqueos, es decir, asumir que nunca van a ocurrir en el sistema.

La figura 1.19 muestra un esquema general con las distintas alternativas existentes a la hora de tratar los interbloqueos. A continuación se discutirán los fundamentos de cada una de estas alternativas, planteando las ventajas y desventajas que tienen.

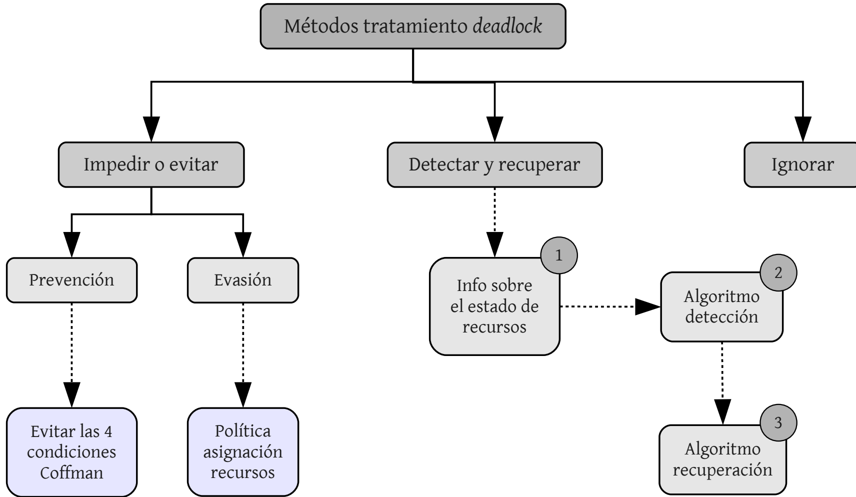


Figura 1.19: Diagrama general con distintos métodos para tratar los interbloqueos.

Con el objetivo de **evitar interbloqueos**, el sistema puede plantear un esquema de prevención o evasión de interbloqueos. Típicamente, las técnicas de prevención de interbloqueos se basan en asegurar que al menos una de las condiciones de *Coffman* no se cumpla. Para ello, estos métodos evitan los interbloqueos limitando el modo en el que los procesos realizan las solicitudes sobre los recursos.

Por otra parte, la evasión de interbloqueos se basa en un esquema más dinámico que gira en torno a la solicitud de más información relativa a las peticiones de recursos por parte de los procesos. Esta idea se suele denominar **política de asignación de recursos**. Un ejemplo típico consiste en especificar el número máximo de instancias que un proceso necesitará para cumplir una tarea.

Si un sistema no usa un mecanismo que evite interbloqueos, otra opción consiste en intentar detectarlos y, posteriormente, tratar de recuperarse de ellos.

En la práctica, todos estos métodos son costosos y su aplicación no supone, generalmente, una gran ventaja con respecto al beneficio obtenido para solventar una situación de interbloqueo. De hecho, los sistemas operativos modernos suelen basarse en la hipótesis de que no se producirán interbloqueos, delegando en el programador esta problemática. Esta aproximación se denomina comúnmente *algoritmo del avestruz*, y es el caso de los sistemas operativos UNIX y Windows.

En determinadas situaciones es deseable ignorar un problema por completo debido a que el beneficio obtenido al resolverlo es menor que la generación de un posible fallo.

Prevención de interbloqueos

Como se ha comentado anteriormente, las técnicas de prevención de interbloqueos se basan en garantizar que algunas de las cuatro condiciones de *Coffman* no se cumplan.

La condición de **exclusión mutua** está asociada a la propia naturaleza del recurso, es decir, depende de si el recurso se puede compartir o no. Los recursos que pueden compartirse no requieren un acceso mutuamente excluyente, como por ejemplo un archivo de sólo lectura. Sin embargo, en general no es posible garantizar que se evitará un interbloqueo incumpliendo la condición de exclusión mutua, ya que existen recursos, como por ejemplo una impresora, que por naturaleza no se pueden compartir..

La condición de **retener y esperar** se puede negar si se garantiza de algún modo que, cuando un proceso solicite un recurso, dicho proceso no esté reteniendo ningún otro recurso. Una posible solución a esta problemática consiste en solicitar todos los recursos necesarios antes de ejecutar una tarea. De este modo, la política de asignación de recursos decidirá si asigna o no todo el bloque de recursos necesarios para un proceso. Otra posible opción sería plantear un protocolo que permitiera a un proceso solicitar recursos si y sólo si no tiene ninguno retenido.

¿Qué desventajas presentan los esquemas planteados para evitar la condición de *retener y esperar*?

Estos dos planteamientos presentan dos desventajas importantes: i) una baja tasa de uso de los recursos, dado que los recursos pueden asignarse y no utilizarse durante un largo periodo de tiempo, y ii) el problema de la inanición, ya que un proceso que necesite recursos muy solicitados puede esperar de forma indefinida.

La condición de **no apropiación** se puede incumplir desalojando los recursos que un proceso retiene en el caso de solicitar otro recurso que no se puede asignar de forma inmediata. Es decir, ante una situación de espera por parte de un proceso, éste liberaría los recursos que retiene actualmente. Estos recursos se añadirían a la lista de recursos que el proceso está esperando. Así, el proceso se reiniciará cuando pueda recuperar sus antiguos recursos y adquirir los nuevos, solucionando así el problema planteado.

Este tipo de protocolos se suele aplicar a recursos cuyo estado pueda guardarse y restaurarse fácilmente, como los registros de la CPU o la memoria. Sin embargo, no se podría aplicar a recursos como una impresora.

Finalmente, la condición de **espera circular** se puede evitar estableciendo una relación de orden completo a todos los tipos de recursos. De este modo, al mantener los recursos ordenados se puede definir un protocolo de asignación de recursos para evitar la espera circular. Por ejemplo, cada proceso sólo puede solicitar recursos en orden creciente de enumeración, es decir, sólo puede solicitar un recurso *mayor* al resto de recursos ya solicitados. En el caso de necesitar varias instancias de un recurso, se solicitarían todas a la vez. Recuerde que respetar la política de ordenación planteada es responsabilidad del programador.

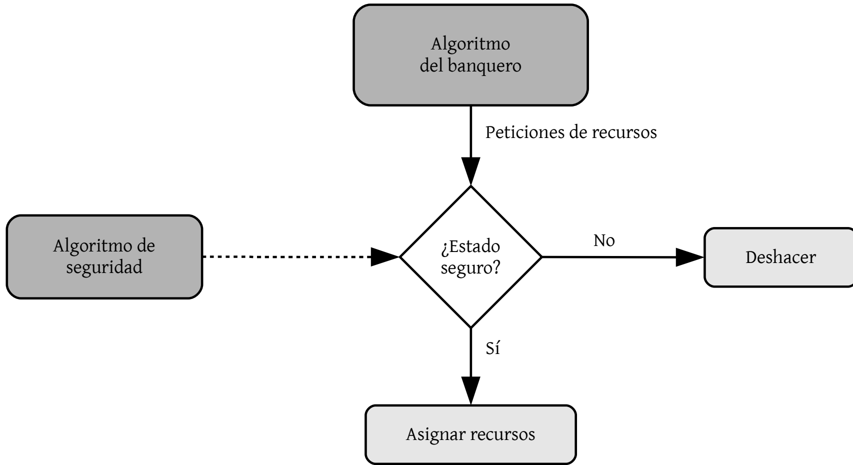


Figura 1.20: Diagrama general de una política de evasión de interbloqueos.

Evasión de interbloqueos

La prevención de interbloqueos se basa en restringir el modo en el que se solicitan los recursos, con el objetivo de garantizar que al menos una de las cuatro condiciones se incumpla. Sin embargo, este planteamiento tiene dos consecuencias indeseables: la baja tasa de uso de los recursos y la reducción del rendimiento global del sistema.

Una posible alternativa consiste en hacer uso de información relativa a cómo se van a solicitar los recursos. De este modo, si se dispone de más información, es posible plantear una política más completa que tenga como meta la evasión de potenciales interbloqueos en el futuro. Un ejemplo representativo de este tipo de información podría ser la secuencia completa de solicitudes y liberaciones de recursos por parte de cada uno de los procesos involucrados.

Las alternativas existentes de evasión de interbloqueos varían en la cantidad y tipo de información necesaria. Una solución simple se basa en conocer únicamente el número máximo de recursos que un proceso puede solicitar de un determinado recurso. El algoritmo de evasión de interbloqueos examinará de manera dinámica el estado de asignación de cada recurso con el objetivo de que no se produzca una espera circular.

En este contexto, el concepto de **estado seguro** se define como aquél en el que el sistema puede asignar recursos a cada proceso (hasta el máximo definido) en un determinado orden sin que se produzca un interbloqueo.

A continuación se describe el denominado **algoritmo del banquero**[3]. En esencia, este algoritmo se basa en que cuando un proceso entra en el sistema, éste ha de establecer el número máximo de instancias de cada tipo de recurso que puede necesitar. Dicho número no puede exceder del máximo disponible en el sistema. La idea general consiste en determinar si la asignación de recursos dejará o no al sistema en un estado seguro. Si es así, los recursos se asigna. Si no lo es, entonces el proceso tendrá que esperar a que otros procesos liberen sus recursos asociados.

La figura 1.20 muestra un diagrama basado en la aplicación de una política de evasión de interbloqueos mediante el algoritmo del banquero y la compración de estados seguros en el sistema.

Para implementar el algoritmo del banquero, es necesario hacer uso de varias estructuras de datos:

- **Available**, un vector de longitud m, definido a nivel global del sistema, que indica el número de instancias disponibles de cada tipo de recurso. Si $available[j] = k$, entonces el sistema dispone actualmente de k instancias del tipo de recurso r_j.
- **Max**, una matriz de dimensiones nxm que indica la demanda máxima de cada proceso. Si $max[i][j] = k$, entonces el proceso p_i puede solicitar, como máximo, k instancias del tipo de recurso r_j.
- **Allocation**, una matriz de dimensiones nxm que indica el número de instancias de cada tipo de recurso asignadas a cada proceso. Si $allocation[i][j] = k$, entonces el proceso p_i tiene actualmente asignadas k instancias del recurso r_j.
- **Need**, una matriz de dimensiones nxm que indica la necesidad restante de recursos por parte de cada uno de los procesos. Si $need[i][j] = k$, entonces el proceso p_i puede que necesite k instancias adicionales del recurso r_j. $need[i][j] = max[i][j] - allocation[i][j]$.

El algoritmo que determina si las solicitudes pueden concederse de manera segura es el siguiente. Sea $request_i$ el vector de solicitud para p_i. Ej. $request_i = [0, 3, 1]$. Cuando p_i hace una solicitud de recursos se realizan las siguientes acciones:

1. Si $request_i \leq need_i$, ir a 2. Si no, generar condición de error, ya que se ha excedido el cantidad máxima de recursos.
2. Si $request_i \leq available$, ir a 3. Si no, p_i tendrá que esperar a que se liberen recursos.
3. Realizar la asignación de recursos.
    a) $available = available - request_i$
    b) $allocation_i = allocation_i + request_i$
    c) $need_i = need_i - request_i$
4. Si el estado resultante es seguro, modificar el estado. Si no, restaurar el estado anterior. p_i esperará a que se le asignen los recursos $request_i$.

[3]El origen de este nombre está en su utilización en sistemas bancarios.

El algoritmo que determina si un estado es seguro es el siguiente. Sean $work$ y $finish$ dos vectores de longitud m y n, inicializados del siguiente modo: $work = available$ y $finish[i] = false \forall i \in \{0, 1, ..., n-1\}$.

1. Hallar i (si no existe, ir a 3) tal que
   a) $finish[i] == false$
   b) $need_i \leq work$
2. $work = work + allocation_i$, $finish[i] = true$, ir a 1.
3. Si $finish[i] == true, \forall i \in \{0, 1, ..., n-1\}$, entonces el sistema está en un estado seguro.

A continuación se plantea un ejercicio sobre la problemática asociada al algoritmo del banquero y a la comprobación de estado seguro. Considere el siguiente estado de asignación de recursos en un sistema:

	Allocation	**Max**
p_0	0 0 1 2	0 0 1 2
p_1	1 0 0 0	1 7 5 0
p_2	1 3 5 4	2 3 5 6
p_3	0 6 3 2	0 6 5 2
p_4	0 0 1 4	0 6 5 6

Así mismo, $available = [1, 5, 2, 0]$

Si se utiliza un mecanismo de evasión del interbloqueo, ¿está el sistema en un estado seguro?

En primer lugar, se hace uso de dos vectores: i) $work = [1, 5, 2, 0]$ y ii) $finish = [f, f, f, f, f]$ y se calcula la necesidad de cada uno de los procesos ($max - allocation$):

- $need_o = [0, 0, 0, 0]$
- $need_1 = [0, 7, 5, 0]$
- $need_2 = [1, 0, 0, 2]$
- $need_3 = [0, 0, 2, 0]$
- $need_4 = [0, 6, 4, 2]$

A continuación, se analiza el estado de los distintos procesos.

- p_0: $need_0 \leq work = [0, 0, 0, 0] \leq [1, 5, 2, 0]; finish[0] = f.$
  - $work = [1, 5, 2, 0] + [0, 0, 1, 2] = [1, 5, 3, 2]$
  - $finish = [v, f, f, f, f]$
- p_1: $need_1 > work = [0, 7, 5, 0] > [1, 5, 3, 2]$

- p_2: $need_2 \leq work = [1,0,0,2] \leq [1,5,3,2]; finish[2] = f$.
  - $work = [1,5,3,2] + [1,3,5,4] = [2,8,8,6]$
  - $finish = [v,f,v,f,f]$
- p_1: $need_1 \leq work = [0,7,5,0] \leq [2,8,8,6]; finish[1] = f$.
  - $work = [2,8,8,6] + [1,0,0,0] = [3,8,8,6]$
  - $finish = [v,v,v,f,f]$
- p_3: $need_3 \leq work = [0,0,2,0] \leq [3,8,8,6]; finish[3] = f$.
  - $work = [3,8,8,6] + [0,6,3,2] = [3,14,11,8]$
  - $finish = [v,v,v,v,f]$
- p_4: $need_4 \leq work = [0,6,4,2] \leq [3,14,11,8]; finish[4] = f$.
  - $work = [3,14,11,8] + [0,0,1,4] = [3,14,12,12]$
  - $finish = [v,v,v,v,v]$

Por lo que se puede afirmar que el sistema se encuentra en un estado seguro.

Si el proceso p_1 solicita recursos por valor de $[0,4,2,0]$, ¿puede atenderse con garantías de inmediato esta petición?

En primer lugar habría que aplicar el algoritmo del banquero para determinar si dicha petición se puede conceder de manera segura.

1. $request_1 \leq need_1 \leftrightarrow [0,4,2,0] \leq [0,7,5,0]$
2. $request_1 \leq available_1 \leftrightarrow [0,4,2,0] \leq [1,5,2,0]$
3. Asignación de recursos.
   a) $available = available - request_1 = [1,5,2,0] - [0,4,2,0] = [1,1,0,0]$
   b) $allocation_1 = allocation_1 + request_1 = [1,0,0,0] + [0,4,2,0] = [1,4,2,0]$
   c) $need_1 = need_1 - request_1 = [0,7,5,0] - [0,4,2,0] = [0,3,3,0]$
4. Comprobación de estado seguro.

En primer lugar, se hace uso de dos vectores: i) $work = [1,1,0,0]$ y ii) $finish = [f,f,f,f,f]$ y se calcula la necesidad de cada uno de los procesos ($max - allocation$):

- $need_o = [0,0,0,0]$
- $need_1 = [0,3,3,0]$
- $need_2 = [1,0,0,2]$
- $need_3 = [0,0,2,0]$
- $need_4 = [0,6,4,2]$

A continuación, se analiza el estado de los distintos procesos.

- p_0: $need_0 \leq work = [0,0,0,0] \leq [1,1,0,0]; finish[0] = f$.
  - $work = [1,1,0,0] + [0,0,1,2] = [1,1,1,2]$
  - $finish = [v,f,f,f,f]$
- p_1: $need_1 > work = [0,3,3,0] > [1,1,1,2]$
- p_2: $need_2 \leq work = [1,0,0,2] \leq [1,1,1,2]; finish[2] = f$.
  - $work = [1,1,1,2] + [1,3,5,4] = [2,4,6,6]$
  - $finish = [v,f,v,f,f]$
- p_1: $need_1 \leq work = [0,3,3,0] \leq [2,4,6,6]; finish[1] = f$.
  - $work = [2,4,6,6] + [1,4,2,0] = [3,8,8,6]$
  - $finish = [v,v,v,f,f]$
- p_3: $need_3 \leq work = [0,0,2,0] \leq [3,8,8,6]; finish[3] = f$.
  - $work = [3,8,8,6] + [0,6,3,2] = [3,14,11,8]$
  - $finish = [v,v,v,v,f]$
- p_4: $need_4 \leq work = [0,6,4,2] \leq [3,14,11,8]; finish[4] = f$.
  - $work = [3,14,11,8] + [0,0,1,4] = [3,14,12,12]$
  - $finish = [v,v,v,v,v]$

Se puede afirmar que el sistema se encuentra en un estado seguro después de la nueva asignación de recursos al proceso p_1.

1.3. Fundamentos de tiempo real

El rango de aplicación de los dispositivos electrónicos, incluyendo los ordenadores, ha crecido exponencialmente en los últimos años. Uno de estos campos está relacionado con las **aplicaciones de tiempo real**, donde es necesario llevar a cabo una determinada funcionalidad atendiendo a una serie de restricciones temporales que son esenciales en relación a los requisitos de dichas aplicaciones. Por ejemplo, considere el sistema de control de un coche. Este sistema ha de ser capaz de responder a determinadas acciones en *tiempo real*.

Es importante resaltar que las aplicaciones de tiempo real tienen características que los hacen particulares con respecto a otras aplicaciones más tradicionales de procesamiento de información. A lo largo del tiempo han surgido herramientas y lenguajes de programación especialmente diseñados para facilitar su desarrollo.

1.3.1. ¿Qué es un sistema de tiempo real?

Según el *Oxford Dictionary of Computing*, un sistema de tiempo real se define como *«cualquier sistema en el que el tiempo en el que se produce la salida es significativo. Esto generalmente es porque la entrada corresponde a algún movimiento en el mundo físico, y la salida está relacionada con dicho movimiento. El intervalo entre el tiempo de entrada y el de salida debe ser lo suficientemente pequeño para una temporalidad aceptable»*.

STR

Los sistemas de tiempo real también se conocen como sistemas empotrados o embebidos, debido a su integración en el propio dispositivo que proporciona una determinada funcionalidad.

El **concepto de temporalidad** resulta esencial en un sistema de tiempo real, ya que marca su diferencia en términos de requisitos con respecto a otro tipo de sistemas.

Como ejemplo representativo, considera la situación en la que un usuario interactúa con un sistema de venta de entradas de cine. Ante una petición de compra, el usuario espera que el sistema responda en un intervalo de tiempo razonable (quizás no más de cinco segundos). Sin embargo, una demora en el tiempo de respuesta por parte del sistema no representa una situación crítica.

Por el contrario, considere el sistema de *airbag* de un coche, el cual está controlado por un microprocesador. Ante una situación de emergencia, como por ejemplo un choque con otro vehículo, el sistema ha de garantizar una **respuesta en un intervalo de tiempo acotado** y perfectamente definido con el objetivo de garantizar una respuesta adecuada del sistema de seguridad. Éste sí es un ejemplo representativo de sistema de tiempo real.

Figura 1.21: Probando un sistema de airbag en un helicóptero OH-58D (fuente Wikipedia).

Desde el punto de vista del diseño, los sistemas de tiempo real se suelen clasificar en **estrictos** (*hard*) y **no estrictos** (*soft*) [3], dependiendo si es absolutamente necesario que la respuesta se produzca antes de un tiempo límite (*deadline*) especificado o no, respectivamente. En el caso de los sistemas de tiempo real no estrictos, el tiempo de respuesta sigue siendo muy importante pero el sistema garantiza su funcionamiento incluso cuando este tiempo límite se incumple ocasionalmente.

Un sistema de tiempo real no ha de ser necesariamente muy rápido, sino que ha de responder en un intervalo de tiempo previamente definido, es decir, ha de mantener un comportamiento determinista.

Tradicionalmente, los sistemas de tiempo real se han utilizado para el control de procesos, la fabricación y la comunicación. No obstante, este tipo de sistemas también se utilizan en un contexto más general con el objetivo de proporcionar una monitorización continua de un determinado entorno físico.

Correctitud STR

La correctitud de un sistema de tiempo real no sólo depende de la correctitud del resultado obtenido, sino también del tiempo empleado para obtenerlo.

En el diseño y desarrollo de sistemas de tiempo real son esenciales aspectos relacionados con la programación concurrente, la planificación en tiempo real y la fiabilidad y la tolerancia a fallos. Estos aspectos serán cubiertos en capítulos sucesivos.

Características de un sistema de tiempo real

El siguiente listado resume las principales características que un sistema de tiempo real puede tener [3]. Idealmente, un lenguaje de programación o sistema operativo que se utilice para el desarrollo de un sistema de tiempo real debería proporcionar dichas características.

- **Complejidad**, en términos de tamaño o número de líneas de código fuente y en términos de variedad de respuesta a eventos del mundo real. Esta característica está relacionada con la mantenibilidad del código.
- **Tratamiento de números reales**, debido a la precisión necesaria en ámbitos de aplicación tradicionales de los sistemas de tiempo real, como por ejemplo los sistemas de control.
- **Fiabilidad y seguridad**, en términos de robustez del software desarrollado y en la utilización de mecanismos o esquemas que la garanticen. El concepto de certificación cobra especial relevancia en el ámbito de la fiabilidad como solución estándar que permite garantizar la robustez de un determinado sistema.
- **Concurrencia**, debido a la necesidad real de tratar con eventos que ocurren de manera paralela en el mundo real. En este contexto, existen lenguajes de programación como Ada que proporcionan un soporte nativo a la concurrencia.
- **Funcionalidad de tiempo real**, debido a la necesidad de trabajar con elementos temporales a la hora de construir un sistema de tiempo real, así como establecer las acciones a realizar ante el incumplimiento de un requisito temporal.
- **Interacción HW**, ya que la propia naturaleza de un sistema empotrado requiere la interacción con dispositivos hardware.
- **Eficiencia**, debido a que la implementación de, por ejemplo, un sistema crítico ha de ser más eficiente que en otro tipo de sistemas.

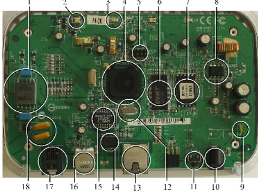


Figura 1.22: Distintos componentes del hardware interno de un módem/-router de ADSL (fuente Wikipedia).

En el capítulo 5 se cubrirán más aspectos de los sistemas de tiempo real. En concreto, se abordará el tema de la **planificación** en sistemas de tiempo real y se discutirán distintos métodos para realizar el cálculo del tiempo de respuesta. Este cálculo es esencial para determinar si un sistema de tiempo real es planificable, es decir, si las distintas tareas que lo componen son capaces de garantizar una respuesta antes de un determinado tiempo límite o *deadline*.

1.3.2. Herramientas para sistemas de tiempo real

El abanico de herramientas existentes para el diseño y desarrollo de sistemas de tiempo real es muy amplio, desde herramientas de modelado, sistemas operativos y lenguajes de programación hasta estándares de diversa naturaleza pasando por compiladores y depuradores.

En el ámbito particular de los **lenguajes de programación** es importante destacar las distintas alternativas existentes, ya sean de más bajo o más alto nivel. Tradicionalmente, una herramienta representativa en el desarrollo de sistemas empotrados ha sido el lenguaje ensamblador, debido a su flexibilidad a la hora de interactuar con el hardware subyacente y a la posibilidad de llevar a cabo implementaciones eficientes. Desafortunadamente, el uso de un lenguaje ensamblador es costoso y propenso a errores de programación.

Los lenguajes de programación de sistemas, como por ejemplo C, representan una solución de más alto nivel que tiene como principal ventaja su amplia utilización en toda la industria del software. No obstante, es necesario el uso de un sistema operativo para dar soporte a los aspectos esenciales de concurrencia y tiempo real.

Así mismo, existen lenguajes de programación de más alto nivel, como por ejemplo Ada, que sí proporcionan un enfoque nativo de concurrencia y tiempo real. En el caso particular de Ada, su diseño estuvo marcado por la necesidad de un lenguaje más adecuado para el desarrollo de sistemas críticos.

LynuxWorks

Un ejemplo representativo de sistema operativo de tiempo real es *LynuxWorks*, el cual da soporte a una virtualización completa en dispositivos empotrados.

En el caso de los **sistemas operativos**, es importante destacar la existencia de algunos sistemas de tiempo real que tienen como principales características el determinismo, con el objetivo de garantizar los tiempos de respuesta de las tareas a las que dan soporte, la integración de la concurrencia en el ámbito del tiempo real y la posibilidad de acceder directamente al hardware subyacente.

Finalmente, en la industria existen diversos **estándares** ideados para simplificar el desarrollo de sistemas de tiempo real y garantizar, en la medida de lo posible, su interoperabilidad. Un ejemplo representativo es POSIX y sus extensiones para tiempo real. Como se discutirá en sucesivos capítulos, el principal objetivo de POSIX es proporcionar una estandarización en las llamadas al sistema de distintos sistemas operativos.