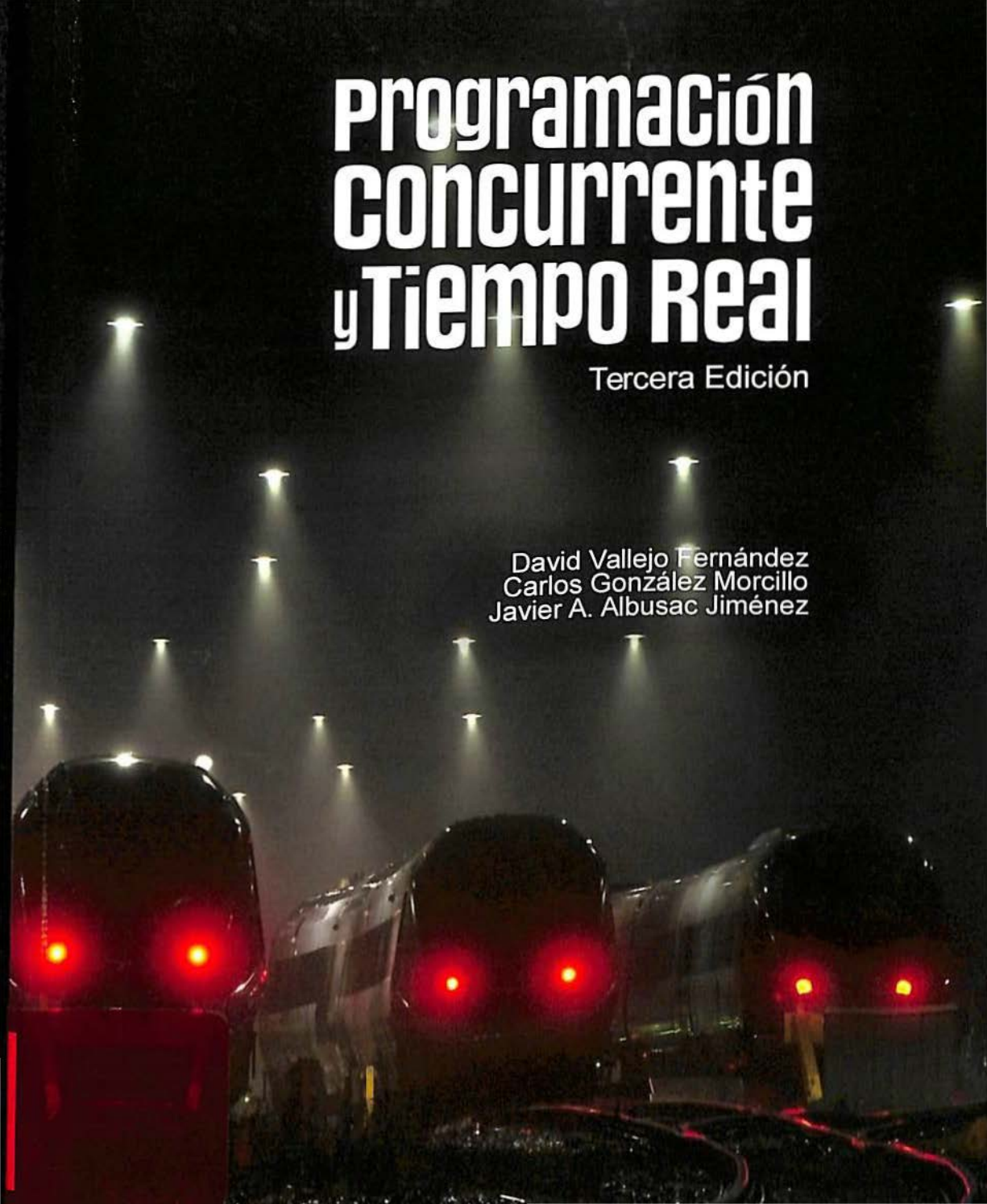
Programación Concurrente y Tiempo Real

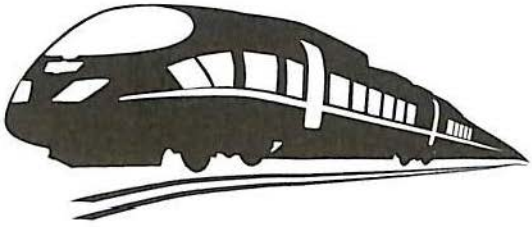
Tercera Edición

David Vallejo Fernández
Carlos González Morcillo
Javier A. Albusac Jiménez



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.

Antes de discutir los semáforos en POSIX, es importante destacar que la definición de semáforo discutida en esta sección requiere una **espera activa**. Mientras un proceso esté en su sección crítica, cualquier otro proceso que intente acceder a la suya deberá ejecutar de manera continuada un bucle en el código de entrada. Este esquema basado en espera activa hace que se desperdicien miles de ciclos de CPU en un sistema multiprogramado, aunque tiene la ventaja de que no se producen cambios de contexto.

Para evitar este tipo de problemática, es posible habilitar el bloqueo de un proceso que ejecuta *wait* sobre un semáforo con valor negativo. Esta operación de bloqueo coloca al proceso en una cola de espera vinculada con el semáforo de manera que el estado del proceso pasa a *en espera*. Posteriormente, el control pasa al planificador de la CPU, que seleccionará otro proceso para su ejecución.

Un proceso bloqueado se reanudará cuando otro proceso ejecuta *signal* sobre el semáforo, cambiando el estado del proceso original de *en espera* a *preparado*.

2.2. Implementación

2.2.1. Semáforos

En esta sección se discuten las **primitivas POSIX** más importantes relativas a la creación y manipulación de semáforos y segmentos de memoria compartida. Así mismo, se plantea el uso de una interfaz que facilite la manipulación de los semáforos mediantes funciones básicas para la creación, destrucción e interacción mediante operaciones *wait* y *signal*.

En POSIX, existe la posibilidad de manejar **semáforos nombrados**, es decir, semáforos que se pueden manipular mediante cadenas de texto, facilitando así su manejo e incrementando el nivel semántico asociado a los mismos.

En el listado de código 2.3 se muestra la interfaz de la primitiva *sem_open()*, utilizada para abrir un semáforo nombrado.

Listado 2.3: Primitiva sem_open en POSIX

```
#include <semaphore.h>

/* Devuelve un puntero al semáforo o SEM FAILED */
sem_t *sem_open (
    const char *name,  /* Nombre del semáforo */
    int oflag,         /* Flags */
    mode_t mode,       /* Permisos */
    unsigned int value /* Valor inicial */
);

sem_t *sem_open (
    const char *name,  /* Nombre del semáforo */
    int oflag,         /* Flags */
);
```

El valor de retorno de la primitiva *sem_open()* es un puntero a una estructura del tipo *sem_t* que se utilizará para llamar a otras funciones vinculadas al concepto de semáforo.

POSIX contempla diversas primitivas para cerrar (*sem_close()*) y eliminar (*sem_unlink()*) un semáforo, liberando así los recursos previamente creados por *sem_open()*. La primera tiene como parámetro un puntero a semáforo, mientras que la segunda sólo necesita el nombre del mismo.

Listado 2.4: Primitivas sem_close y sem_unlink en POSIX

```
#include <semaphore.h>

/* Devuelven 0 si todo correcto o -1 en caso de error */
int sem_close  (sem_t *sem);
int sem_unlink (const char *name);
```

Finalmente, las primitivas para decrementar e incrementar un semáforo se exponen a continuación. Note cómo la primitiva de incremento utiliza el nombre *post* en lugar del tradicional *signal*.

Listado 2.5: Primitivas sem_wait y sem_post en POSIX

```
#include <semaphore.h>

/* Devuelven 0 si todo correcto o -1 en caso de error */
int sem_wait (sem_t *sem);
int sem_post (sem_t *sem);
```

En el presente capítulo se utilizarán los semáforos POSIX para codificar soluciones a diversos problemas de sincronización que se irán planteando en la sección 2.3. Para facilitar este proceso se ha definido una **interfaz** con el objetivo de simplificar la manipulación de dichos semáforos, el cual se muestra en el listado 2.6.

Listado 2.6: Interfaz de gestión de semáforos

```
#include <semaphore.h>

typedef int bool;
#define false 0
#define true 1

/* Crea un semáforo POSIX */
sem_t *crear_sem (const char *name, unsigned int valor);

/* Obtener un semáforo POSIX (ya existente) */
sem_t *get_sem (const char *name);

/* Cierra un semáforo POSIX */
void destruir_sem (const char *name);

/* Incrementa el semáforo */
void signal_sem (sem_t *sem);

/* Decrementa el semáforo */
void wait_sem (sem_t *sem);
```

Patrón fachada

La interfaz de gestión de semáforos propuesta en esta sección es una implementación del patrón fachada *facade*.

El diseño de esta interfaz se basa en utilizar un esquema similar a los semáforos nombrados en POSIX, es decir, identificar a los semáforos por una cadena de caracteres. Dicha cadena se utilizará para crear un semáforo, obtener el puntero a la estructura de tipo *sem_t* y destruirlo. Sin embargo, las operaciones *wait* y *signal* se realizarán sobre el propio puntero a semáforo.

2.2.2. Memoria compartida

Los problemas de sincronización entre procesos suelen integrar algún tipo de recurso compartido cuyo acceso, precisamente, hace necesario el uso de algún mecanismo de sincronización como los semáforos. El recurso compartido puede ser un **fragmento de memoria**, por lo que resulta relevante estudiar los mecanismos proporcionados por el sistema operativo para dar soporte a la memoria compartida. En esta sección se discute el soporte POSIX para memoria compartida, haciendo especial hincapié en las primitivas proporcionadas y en un ejemplo particular.

La gestión de memoria compartida en POSIX implica la **apertura o creación** de un segmento de memoria compartida asociada a un nombre particular, mediante la primitiva *shm_open()*. En POSIX, los segmentos de memoria compartida actúan como archivos en memoria. Una vez abiertos, es necesario establecer el tamaño de los mismos mediante *ftruncate()* para, posteriormente, mapear el segmento al espacio de direcciones de memoria del usuario mediante la primitiva *mmap*.

A continuación se expondrán las primitivas POSIX necesarias para la gestión de segmentos de memoria compartida. Más adelante, se planteará un ejemplo concreto para compartir una variable entera.

Listado 2.7: Primitivas shm_open y shm_unlink en POSIX

```
1  #include <mman.h>
2
3  /* Devuelve el descriptor de archivo o -1 si error */
4  int shm_open(
5     const char *name, /* Nombre del segmento */
6     int oflag,        /* Flags */
7     mode_t mode       /* Permisos */
8  );
9
10 /* Devuelve 0 si todo correcto o -1 en caso de error */
11 int shm_unlink(
12    const char *name /* Nombre del segmento */
13 );
```

Recuerde que, cuando haya terminado de usar el descriptor del archivo, es necesario utilizar *close()* como si se tratara de cualquier otro descriptor de archivo.

Listado 2.8: Primitiva close

```
#include <unistd.h>

int close(
    int fd /* Descriptor del archivo */
);
```

Después de crear un objeto de memoria compartida es necesario establecer de manera explícita su **longitud** mediante *ftruncate()*, ya que su longitud inicial es de cero bytes.

Listado 2.9: Primitiva ftruncate

```
#include <unistd.h>
#include <sys/types.h>

int ftruncate(
    int fd,      /* Descriptor de archivo */
    off_t length /* Nueva longitud */
);
```

A continuación, se ***mapea*** el objeto al espacio de direcciones del usuario mediante *mmap()*. El proceso inverso se realiza mediante la primitiva *munmap()*. La función *mmap()* se utiliza para la creación, consulta y modificación de valor almacena en el objeto de memoria compartida.

Listado 2.10: Primitiva mmap

```
#include <sys/mman.h>

void *mmap(
  void *addr,    /* Dirección de memoria */
  size_t length, /* Longitud del segmento */
  int prot,      /* Protección */
  int flags,     /* Flags */
  int fd,        /* Descriptor de archivo */
  off_t offset   /* Desplazamiento */
);

int munmap(
  void *addr,   /* Dirección de memoria */
  size_t length /* Longitud del segmento */
);
```

Compartiendo un valor entero

En función del dominio del problema a resolver, el segmento u objeto de memoria compartida tendrá más o menos longitud. Por ejemplo, sería posible definir una estructura específica para un **tipo abstracto de datos** y crear un segmento de memoria compartida para que distintos procesos puedan modificar dicho segmento de manera concurrente. En otros problemas sólo será necesario utilizar un valor entero a compartir entre los procesos. La creación de un segmento de memoria para este último caso se estudiará a continuación.

Al igual que en el caso de los semáforos, se ha definido una **interfaz** para simplicar la manipulación de un valor entero compartido. Dicha interfaz facilita la creación, destrucción, modificación y consulta de un valor entero que puede ser compartido por múltiples procesos. Al igual que en el caso de los semáforos, la manipulación del valor entero se realiza mediante un esquema basado en nombres, es decir, se hacen uso de cadenas de caracteres para identificar los segmentos de memoria compartida.

Listado 2.11: Interfaz de gestión de un valor entero compartido

```
1  /* Crea un objeto de memoria compartida */
2  /* Devuelve el descriptor de archivo */
3  int crear_var (const char *name, int valor);
4
5  /* Obtiene el descriptor asociado a la variable */
6  int obtener_var (const char *name);
7
8  /* Destruye el objeto de memoria compartida */
9  void destruir_var (const char *name);
10
11 /* Modifica el valor del objeto de memoria compartida */
12 void modificar_var (int shm_fd, int valor);
13
14 /* Devuelve el valor del objeto de memoria compartida */
15 void consultar_var (int shm_fd, int *valor);
```

Resulta interesante resaltar que la función *consultar* almacena el valor de la consulta en la variable *valor*, pasada por referencia. Este esquema es más eficiente, evita la generación de copias en la propia función y facilita el proceso de *unmapping*.

En el listado de código 2.12 se muestra la implementación de la función *crear_var* cuyo objetivo es el de crear e inicializar un segmento de memoria compartida para un valor entero. Note cómo es necesario llevar a cabo el *mapping* del objeto de memoria compartida con el objetivo de asignarle un valor (en este caso un valor entero).

Posteriormente, es necesario establecer el tamaño de la variable compartida, mediante la función *ftruncate*, asignar de manera explícita el valor y, finalmente, realizar el proceso de *unmapping* mediante la función *munmap*.

Listado 2.12: Creación de un segmento de memoria compartida

```
1  int crear_var (const char *name, int valor) {
2    int shm_fd;
3    int *p;
4
5    /* Abre el objeto de memoria compartida */
6    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0644);
7    if (shm_fd == -1) {
8      fprintf(stderr, "Error al crear la variable: %s\n", strerror(errno));
9      exit(EXIT_FAILURE);
10   }
11
12   /* Establecer el tamaño */
13   if (ftruncate(shm_fd, sizeof(int)) == -1) {
14     fprintf(stderr, "Error al truncar la variable: %s\n", strerror(errno));
15     exit(EXIT_FAILURE);
16   }
17
18   /* Mapeo del objeto de memoria compartida */
19   p = mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
20   if (p == MAP_FAILED) {
21     fprintf(stderr, "Error al mapear la variable: %s\n", strerror(errno));
22     exit(EXIT_FAILURE);
23   }
24
25   *p = valor;
26   munmap(p, sizeof(int));
27
28   return shm_fd;
29 }
```

2.3. Problemas clásicos de sincronización

En esta sección se plantean algunos problemas clásicos de sincronización y se discuten posibles soluciones basadas en el uso de semáforos. En la mayor parte de ellos también se hace uso de segmentos de memoria compartida para compartir datos.

2.3.1. El buffer limitado

El problema del buffer limitado ya se presentó en la sección 1.2.1 y también se suele denominar el *problema del productor/consumidor*. Básicamente, existe un espacio de almacenamiento común limitado, es decir, dicho espacio consta de un conjunto finito de huecos que pueden contener o no elementos. Por una parte, los productores insertarán elementos en el buffer. Por otra parte, los consumidores los extraerán.

La primera cuestión a considerar es que se ha de controlar el acceso exclusivo a la **sección crítica**, es decir, al propio buffer. La segunda cuestión importante reside en controlar **dos situaciones**: i) no se puede insertar un elemento cuando el buffer está lleno y ii) no se puede extraer un elemento cuando el buffer está vacío. En estos dos casos, será necesario establecer un mecanismo para bloquear a los procesos correspondientes.