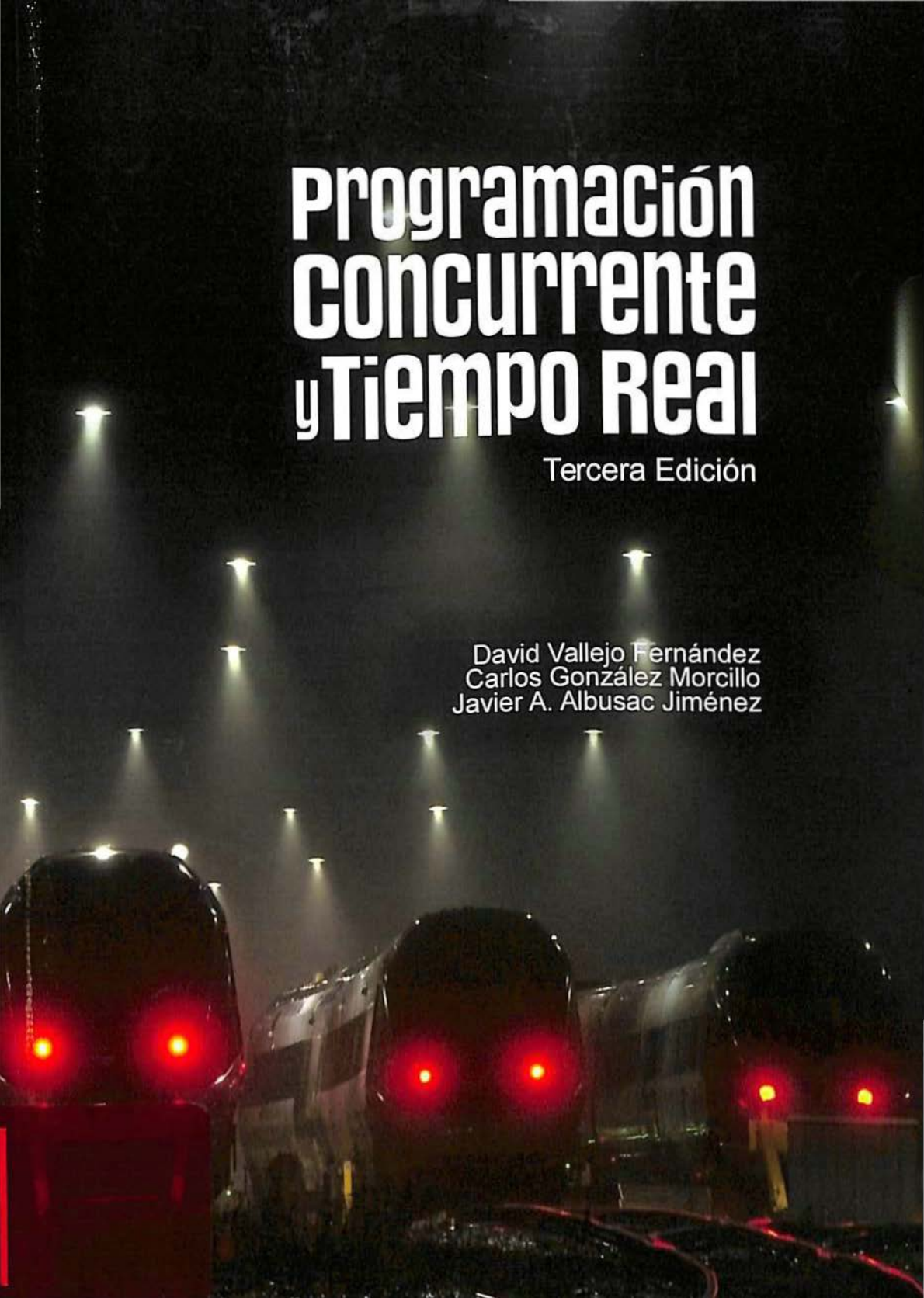
Programación concurrente y Tiempo Real

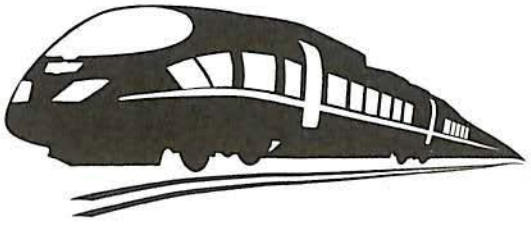
Tercera Edición

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

(SISTEMAS OPERATIVOS)
(LENGUAJES DE PROGRAMACIÓN)
(PROGRAMACIÓN DE COMPUTADORES)
(HARDWARE)
(POSIX) (C) (C++)

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.

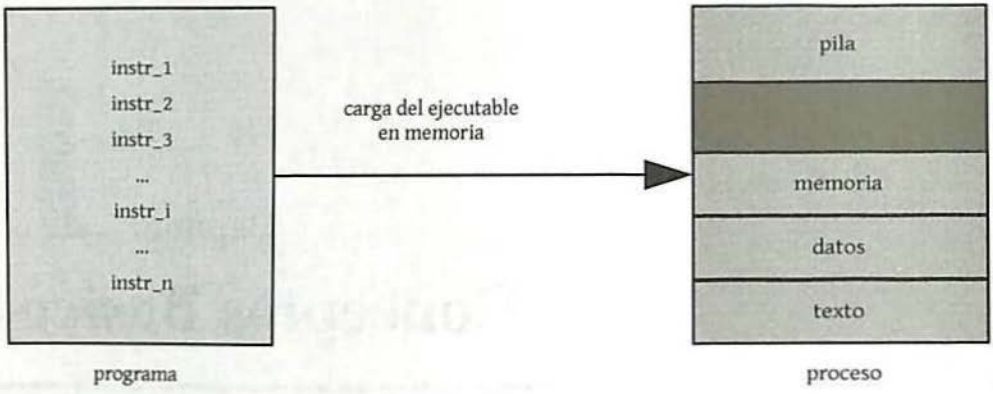


Figura 1.1: Esquema gráfico de un programa y un proceso.

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:

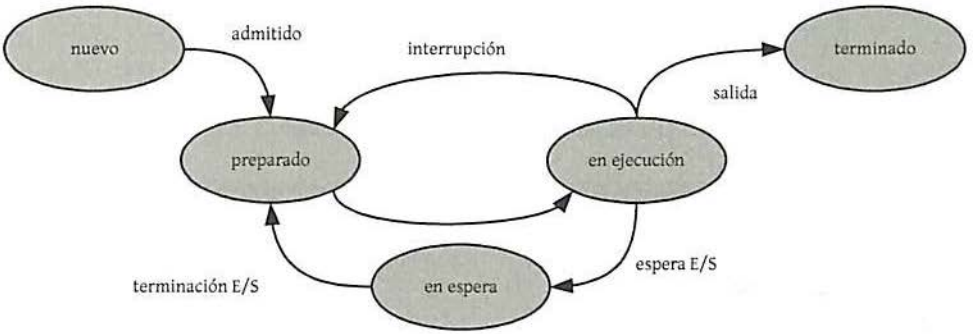


Figura 1.2: Diagrama general de los estados de un proceso.

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.

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**.

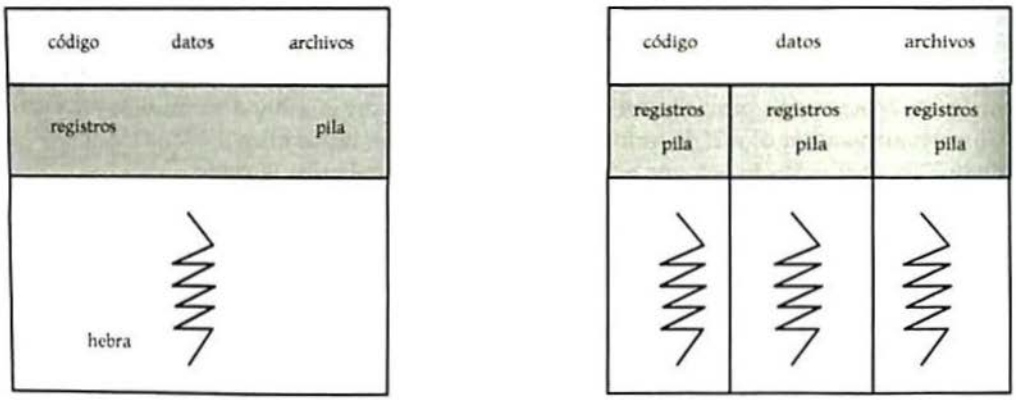


Figura 1.7: Esquema gráfico de los modelos monohilo y multihilo.

- 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.

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*.

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.

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

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  int suma;
6  void *mi_hilo (void *valor);
7
8  int main (int argc, char *argv[]) {
9    pthread_t tid;
10   pthread_attr_t attr;
11
12   if (argc != 2) {
13     fprintf(stderr, "Uso: ./pthread <entero>\n");
14     return -1;
15   }
16
17   pthread_attr_init(&attr); // Att predeterminados.
18   // Crear el nuevo hilo.
19   pthread_create(&tid, &attr, mi_hilo, argv[1]);
20   pthread_join(tid, NULL);  // Esperar finalización.
21
22   printf("Suma total: %d.\n", suma);
23
24   return 0;
25 }
26
27 void *mi_hilo (void *valor) {
28   int i, ls;
29   ls = atoi(valor);
30   i = 0, suma = 0;
31
32   while (i <= ls) {
33     suma += (i++);
34   }
35
36   pthread_exit(0);
37 }
```

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.