

# Fundamentos de los Sistemas Operativos

Departamento de Informática de Sistemas y Computadoras (DISCA)

*Universitat Politècnica de València*



## Práctica 5

### Creación y Sincronización de Hilos POSIX (v2.0)

#### Contenido

1. Objetivos .....	3
2. Creación de hilos .....	3
Ejercicio 1: Trabajando con <code>pthread_join</code> y <code>pthread_exit</code> .....	5
3. Variables compartidas entre hilos .....	5
4. Observando condiciones de carrera .....	7
Ejercicio2: Creación de hilos “CondCarr.c” .....	7
5. Ejecución secuencial .....	8
Ejercicio3: Ejecución secuencial “CondCarrHS.c” .....	8
6. Soluciones para evitar la condición de carrera .....	9
7. Protegiendo sección crítica .....	9
Ejercicio4: Solución de sincronización con <code>test_and_set</code> .....	9
Ejercicio5: Solución de sincronización con semáforos .....	11
Ejercicio6: Solución de sincronización con mutex .....	11
8. Anexos .....	13
Anexo 1: Código fuente de apoyo, “CondCarr.c” .....	13
Anexo 2: Sincronización por espera activa. <code>test_and_set</code> .....	14
Anexo 3: Sincronización por espera pasiva. Semáforos .....	15
Anexo 4: Sincronización por espera pasiva. “Mutex de pthread” .....	17



## 1. Objetivos

- **Adquirir experiencia en el manejo de funciones estándar POSIX para creación y espera de hilos.**
- Trabajar con un escenario donde se produzcan operaciones concurrentes.
- **Comprender cuándo se producen condiciones de carrera**, así como los mecanismos más básicos para evitar este problema.
- Trabajar soluciones al problema de condición de carrera con **espera activa** y **espera pasiva**.

## 2. Creación de hilos

El código de la figura-1 constituye el esqueleto básico de una función que utiliza hilos en su implementación.

```
/**
 * Programa de ejemplo "Hola mundo" con pthreads.
 * Para compilar teclea: gcc hola.c -lpthread -o hola
 */
#include <stdio.h>
#include <pthread.h>
#include <string.h>
#include <unistd.h>

void *Imprime( void *ptr )
{
    char *men;
    men=(char*)ptr;

    //EJERCICIO1.b
    write(1,men,strlen(men));
}
int main()
{
    pthread_attr_t atrib;
    pthread_t hilo1, hilo2;

    pthread_attr_init( &atrib );

    pthread_create( &hilo1, &atrib, Imprime, "Hola \n");
    pthread_create( &hilo2, &atrib, Imprime, "mundo \n");

    //EJERCICIO1.a
    pthread_join( hilo1, NULL);
    pthread_join( hilo2, NULL);

    return 0;
}
```

Figura-1: Esqueleto básico de un programa con hilos POSIX.

Cree un archivo “hola.c” que contenga dicho código, compílelo y ejecútelo desde la línea de órdenes.

```
$ gcc hola.c -lpthread -o hola
```

Como se observa en el código de la figura-1, las novedades que introduce el manejo de hilos van de la mano de las funciones necesarias para inicializarlos, de las cuales sólo hemos hecho uso de las más básicas o imprescindibles.

- Tipos **pthread\_t** y **pthread\_attr\_t** que se acceden desde el archivo de cabecera **pthread.h**.

```
#include <pthread.h >
pthread_t  th;
pthread_attr_t attr;
```

- Función **pthread\_attr\_init** encargada de asignar unos valores por defecto a los elementos de la estructura de atributos de un hilo. ¡AVISO! Si no se inicializan los atributos, el hilo no se puede crear

```
#include <pthread.h >

int pthread_attr_init(pthread_attr_t *attr)
```

- Función **pthread\_create** encargada de crear un hilo.

```
#include <pthread.h >

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
void *(*start_routine) (void *), void *arg);
```

#### Parámetros de **pthread\_create**:

- **thread**: Contendrá el identificador del hilo.
- **attr**: Especifica los atributos del hilo. Con valor NULL, reciben valores por defecto: *“the created thread is joinable (not detached) and has default (non real-time) scheduling policy”*.
- **start\_routine**: Función que define el comportamiento del hilo que se está creando.
- **arg**: Argumento que se pasa a la función del hilo (**start\_routine**) y cuyo uso dependerá de cada función.

#### Valor de retorno de la función **pthread\_create**:

Devuelve 0, si la función se ejecuta con éxito. En caso de error, la función devuelve un valor distinto de cero.

- Función **pthread\_join**. Su efecto es suspender al hilo que la invoca hasta que el hilo que se le especifica como parámetro termine. Este comportamiento es necesario ya que cuando el hilo principal “termina” destruye el proceso y, por lo tanto, obliga a la terminación de todos los hilos que se hayan creado.

#### Parámetros de **pthread\_join**:

- **thread**: Parámetro que identifica al hilo a esperar.
- **exit\_status**: Contiene el valor que el hilo terminado comunica al hilo que invoca a **pthread\_join**.

```
#include <pthread.h >

int pthread_join(pthread_t thread, void **exit_status);
```

- Función **pthread\_exit** permite a un hilo terminar voluntariamente su ejecución. La finalización del último hilo de un proceso finaliza el proceso. Mediante el parámetro **exit\_status** puede comunicar un valor de terminación a otro hilo que estuviera esperando su finalización.

```
#include <pthread.h >

int pthread_exit(void *exit_status);
```

### Ejercicio 1: Trabajando con `pthread_join` y `pthread_exit`

Compruebe el comportamiento de la llamada `pthread_join` realizando las modificaciones que se plantean en las siguientes cuestiones en el código del programa “hola.c” mostrado anteriormente.

#### Cuestiones Ejercicio 1:

Elimine (o comente) las llamadas `pthread_join` del hilo principal.

- ¿Qué ocurre? ¿Por qué?

Que los hilos no esperan a que termine el principal

Sustituya las llamadas `pthread_join` por una llamada `pthread_exit(0)`, cerca del punto del programa marcado como `//EJERCICIO1.a`)

- ¿Completa ahora correctamente el programa su ejecución? ¿Por qué?

Si, porque el hilo devuelve un valor de salida y se puede ver la ejecución de la función

Elimine (o comente) cualquier llamada a `pthread_join` o `pthread_exit` (cerca del comentario `//EJERCICIO1.a`) e introduzca en ese mismo punto un retraso de 1 segundo (**usando la función `usleep(...)` de la librería `<unistd.h>` y cuyo definición se muestra a continuación**)

```
#include <unistd.h>
void usleep(unsigned long usec); // usec en microsegundos
```

- ¿Qué ocurre tras la realización de las modificaciones propuestas?

Que ahora también hay suficiente tiempo

Introduzca ahora un retraso de 2 segundos cerca del comentario `//EJERCICIO1.b`

- ¿Qué ocurre ahora? ¿Por qué?

Que al llamar a la función hay 2 segundos de espera, mientras que el main solo tiene 1 segundo, y no da tiempo

### 3. Variables compartidas entre hilos

Para observar la problemática de utilizar variables compartidas, proponemos un problema sencillo en el que dos hilos requieren acceder a una variable compartida **V**. Un hilo `agrega()` que incrementa la variable y otro `resta()` que decrementa la variable. El valor inicial de **V** es de 100, y se realizan los mismos incrementos que decrementos, por lo que tras la ejecución la variable **V** debería valer 100. Para visualizar los valores de la variable **V**, se utiliza un tercer hilo `inspecciona()` que consulta el valor de **V** y lo muestra por pantalla a intervalos de un segundo.

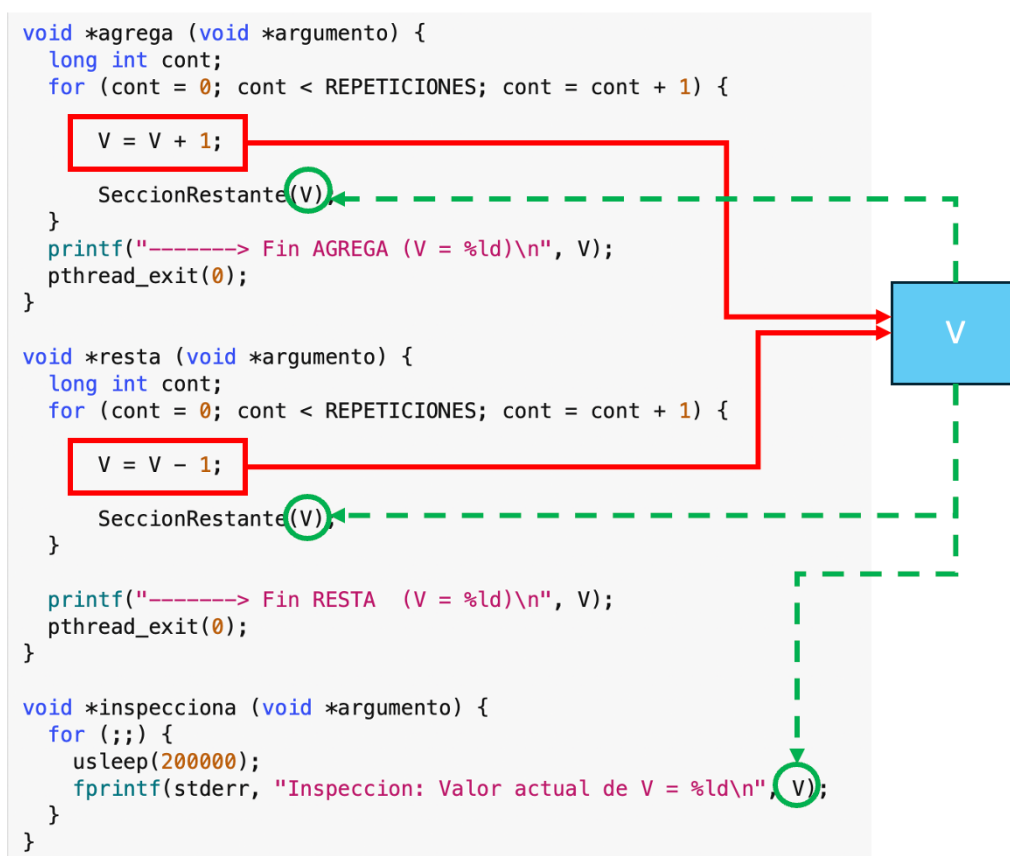


Figura-2: Código de las funciones `agrega`, `resta` e `inspecciona`.

El hilo `inspecciona()` accede a `V` para leer su valor, pero no escribe sobre ella, por lo que no provoca condiciones de carrera. Los hilos `agrega()` y `resta()` acceden a `V` leyéndola y modificándola repetidamente. En detalle, en estos dos hilos, se puede observar que hay un bucle `for` que se ejecuta `REPETICIONES` veces, donde en cada iteración se modifica la variable `V` y luego se ejecuta una función `SeccionRestante(V)` que trabaja con esa variable `V` pero sin modificarla. La operación incremento,  $V=V+1$ , lee la variable, incrementa su valor y escribe en memoria el nuevo valor. Si durante la operación de incremento se intercala la de decremento  $V=V-1$  debido a un cambio de contexto o a la ejecución concurrente de incremento y decremento en núcleos diferentes del procesador, es posible que se produzca una condición de carrera y la variable `V` tome valores inesperados. Es decir, que al finalizar ambos hilos el valor de `V` no sea el inicial, 100 en nuestro caso.

Los escenarios en los que se puede producir la condición de carrera varían según las características de la máquina donde se trabaja, como muestra la Figura-3. Por ejemplo, en un procesador con múltiples núcleos de ejecución (*multi-core*), se podrá observar la condición de carrera fácilmente con valores relativamente bajos de la constante `REPETICIONES` (Figura-2). Si el computador tiene un solo núcleo de ejecución, es menos probable que se produzca una condición de carrera. En este caso habrá que aumentar el número de `REPETICIONES`.

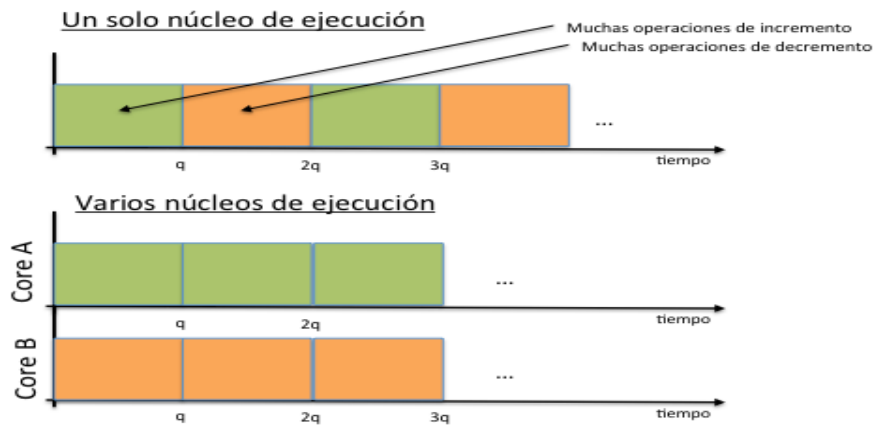


Figura-3: Ejecución de los hilos agrega y resta en CPU, en máquinas con uno y dos núcleos.

## 4. Observando condiciones de carrera

Descargue el material de esta práctica del PoliformaT de la asignatura, donde encontrará un archivo C que contiene el programa que aparece en el Anexo-1 de este boletín. Para compilarlo teclee:

```
$ gcc CondCarr.c -lpthread -o CondCarr
```

### Ejercicio2: Creación de hilos “CondCarr.c”

**Complete el código proporcionado** en “CondCarr.c” de forma que se creen tres hilos: uno ejecutará la función `agrega()`, otro la función `resta()`, y el último hilo ejecutará la función `inspecciona()`, cuyo código se puede observar en la Figura-2. Utilice las llamadas `pthread_attr_init()`, `pthread_create()`, y `pthread_join()`.

Observe que el hilo `inspecciona()` consiste en un bucle infinito y si en la función `main()` hacemos `pthread_join()` sobre él, el programa nunca terminará. Por lo tanto, preste especial atención y asegúrese de hacer `pthread_join()` **sólo para los hilos `agrega()` y `resta()`** ya que el programa debe acabar cuando los hilos `agrega()` y `resta()` terminen.

Compile y ejecute el código implementado. Observe el valor de **V** y determine de forma justificada si se ha producido una condición de carrera o no.

Se ha producido una condición de carrera, ya que :  
-----> VALOR FINAL: V = -9816

La orden **time** muestra el tiempo que tarda en ejecutarse un programa. Ejecute:

```
$ time ./CondCarr
```

**time** muestra el tiempo real (como si cronometrásenos) y los tiempos de CPU (medidos por el planificador) ejecutando instrucciones de usuario y del sistema operativo.

Con la ayuda de la orden **time** averigüe el tiempo de ejecución del programa “CondCarr.c” con condiciones de carrera, ya que no se ha protegido la sección crítica. Anote los tiempos mostrados

./CondCarr 28,64s user 0,00s system 199% cpu 14,382 total

CondCarr.c Sin proteger la sección crítica	
Tiempo real de ejecución	
Tiempo de ejecución en modo usuario	
Tiempo de ejecución en modo sistema	

## 5. Ejecución secuencial

Para evitar la condición de carrera podemos forzar la ejecución secuencial de los hilos `agrega()` y `resta()`. Esto no es una solución a la condición de carrera, es simplemente una forma de evitar el problema.

### Ejercicio3: Ejecución secuencial “CondCarrHS.c”

Cree una copia del fuente modificado en la sección anterior (“CondCarr.c”) y llámelo “CondCarrHS.c”.

La idea es que primero se ejecute el `hiloSuma` y cuando éste termine el `hiloResta`. Modifique el `main` de “CondCarrHS.c” para que cree primero el `hiloInspeccion`, después el `hiloSuma`, espere por este hilo, cree el `hiloResta` y finalmente espere por éste.

Compile “CondCarrHS.c” y ejecute el código implementado. Observe el valor de **V** y determine de forma justificada si se ha producido una condición de carrera o no.

Ahora no se produce ninguna condición de carrera,  
-----> VALOR FINAL: V = 100

Al igual que en la sección anterior, usando la orden **time** muestra el tiempo que tarda en ejecutarse un programa. Ejecute:

```
$ time ./CondCarrHS
```

Anote los tiempos mostrados

./CondCarrHS 24,53s user 0,01s system 99% cpu 24,623 total

CondCarrHS.c Secuencial	
Tiempo real de ejecución	
Tiempo de ejecución en modo usuario	
Tiempo de ejecución en modo sistema	

Comparado con los tiempos medidos anteriormente (ver Tabla “CondCarr.c”), se puede ver que el tiempo real de ejecución casi se ha duplicado. ¿Cuál es la razón?

Porque primero se crea el hilo de suma y se ejecuta la función de resta y la de inspección y luego igual con la resta

Como se puede observar, al ejecutar de forma secuencial los hilos, se ha aumentado el tiempo de ejecución real, y no sacamos provecho de la posible concurrencia en la ejecución de los hilos.



## 6. Soluciones para evitar la condición de carrera

Para evitar condiciones de carrera, es necesario sincronizar el acceso a las **secciones críticas del código**, en nuestro caso las operaciones de decremento e incremento sobre **V**. Esta sincronización debe garantizar la “**exclusión mutua**”, cumpliéndose que mientras un hilo está ejecutando una sección crítica otro hilo no pueda ejecutar simultáneamente su sección crítica. Para conseguir esto, protegeremos las secciones críticas con unas secciones de código como protocolo de entrada y salida, tal como indica la Figura-4.

```
void *agrega (void *argumento) {
    long int cont;

    for (cont = 0; cont < REPETICIONES; cont = cont + 1) {
        Protocolo de Entrada o Sección de Entrada
            V = V + 1;
        Protocolo de Salida o Sección de Salida

        SeccionRestante(V);
    }
    printf("-----> Fin AGREGA (V = %ld)\n", V);
    pthread_exit(0);
}
```

Figura-4: Protocolo de entrada y protocolo de salida a la sección crítica de `agrega()`.

El código del protocolo de entrada y salida dependerá del método de sincronización. En esta práctica estudiaremos tres métodos de sincronización:

- Sincronización mediante espera activa usando la función `test_and_set`.
- Sincronización con espera pasiva, estudiaremos los dos mecanismos que ofrece POSIX:
  - Semáforos: variables de tipo “`sem_t`” en POSIX.
  - Objetos mutex de la biblioteca `pthread`.

**¡AVISO!** El anexo de este documento incluye una descripción detallada sobre las soluciones, que se trabajan en esta práctica, para evitar las condiciones de carrera. Es recomendable que el alumno lea detenidamente dicho anexo antes de desarrollar las actividades.

## 7. Protegiendo sección crítica

Trabaje únicamente sobre la versión del código en la que Sí se observan condiciones de carrera (“`CondCarr.c`”). En los siguientes ejercicios, modificaremos el código protegiendo la sección crítica para comprobar que no se producen condiciones de carrera. También mediremos los tiempos de ejecución de las diferentes versiones y determinar el coste en tiempo de ejecución que implica el uso de exclusión mutua en el acceso a la sección crítica.

### Ejercicio4: Solución de sincronización con `test_and_set`

Una vez comprobado que se producen condiciones de carrera copie el archivo “`CondCarr.c`” en “`CondCarrT.c`”. Modifique el código “`CondCarrT.c`” para garantizar que el acceso a la variable compartida **V** es en exclusión mutua. Para ello realice lo siguiente:

1. Identifique la parte del código correspondiente a sección crítica (S.C), protéjalo con la función `test_and_set`, siguiendo el esquema de la Figura-4 y la Tabla 3 (Anexo 2). Ejecute el programa y compruebe que no se producen condiciones de carrera.

./CondCarrT 28,92s user 0,00s system 198% cpu 14,535 total

2. Utilice el comando **time** para conocer el tiempo de ejecución del programa con la sección crítica protegida y anótelos en la siguiente tabla:

CondCarrT.c Protegiendo la sección crítica con <code>test_and_set</code>	
Tiempo real de ejecución	
Tiempo de ejecución en modo usuario	
Tiempo de ejecución en modo sistema	

3. Compare estos tiempos con los de "CondCarr.c". ¿Hay alguna diferencia significativa? Con respecto a la ejecución secuencial ("CondCarrHS"), ¿qué hemos conseguido?

No hay mucha diferencia significativa

Hemos conseguido que, sin modificar la creación de los hilos, no hayan condiciones de carrera

4. Copie "CondCarrT.c" en "CondCarrTB.c". Observe en "CondCarrTB.c" qué ocurre si reescribe las secciones de entrada y salida y las sitúa en los lugares indicados en la Figura-5

```
void *agrega (void *argumento) {
    long int cont;

    Protocolo de Entrada
    for (cont = 0; cont < REPETICIONES; cont = cont + 1) {

        V = V + 1;

        SeccionRestante(V);

    }
    Protocolo de Salida
    printf("-----> Fin AGREGA (V = %ld)\n", V);
    pthread_exit(0);
}
```

Figura 5: Nuevo lugar de colocación de los protocolos de protección.

5. Anote los resultados de tiempo de ejecución de en la siguiente tabla:

CondCarrTB.c Protegiendo todo el bucle <code>for</code> con <code>test_and_set</code>	
Tiempo real de ejecución	
Tiempo de ejecución en modo usuario	
Tiempo de ejecución en modo sistema	

./CondCarrTB 39,49s user 0,00s system 156% cpu 25,222 total

6. ¿Qué ha ocurrido en este caso? ¿Por qué se ha incrementado también el tiempo de ejecución en modo usuario?

Porque ahora al proteger todo el bucle lo que hace es que tiene que entrar una y otra vez en él, y no solo en la sección crítica, que es la que produce la condición de carrera

### Ejercicio5: Solución de sincronización con semáforos

Copie el archivo "CondCarr.c" en "CondCarrS.c" y realice las modificaciones sobre este último.

1. Proteja la sección crítica utilizando un semáforo POSIX (`sem_t`) como se describe en la Tabla 5 (Anexo 3). Ejecute el programa y compruebe que no se producen condiciones de carrera.
2. Vuelva a ejecutar el código con la orden **time** para obtener el tiempo de ejecución y anote los resultados.  
./CondCarrS 29,51s user 0,03s system 199% cpu 14,840 total

CondCarrS.c	Protegiendo la sección crítica con semáforos <code>sem_t</code>
Tiempo real de ejecución	
Tiempo de ejecución en modo usuario	
Tiempo de ejecución en modo sistema	

3. ¿Qué ha ocurrido en este caso? Compara los resultados con los de `test_and_set`.

Que son muy parecidos, ya que la solución a la condición de carrera es prácticamente idéntica

### Ejercicio6: Solución de sincronización con mutex

Copie del archivo "CondCarr.c" sobre "CondCarrM.c" y realice las modificaciones sobre este último.

1. Proteja la sección crítica utilizando un mutex (`pthread_mutex_t`) tal y como se describe en la Tabla 6 (Anexo 4). Ejecute el nuevo programa y compruebe que no se producen condiciones de carrera.
2. Utilizando la orden `time` ejecute el código para conocer el tiempo de ejecución del programa y anote los resultados en la siguiente tabla.

CondCarrM.c	Protegiendo la sección crítica con mutex de "pthreads"
Tiempo real de ejecución	
Tiempo de ejecución en modo usuario	
Tiempo de ejecución en modo sistema	

./CondCarrM 30,17s user 0,16s system 197% cpu 15,331 total

3. Razone y conteste a las siguientes preguntas.

En el ejemplo desarrollado en esta práctica ¿qué es más eficiente la espera activa o la pasiva?

En general, ¿En qué condiciones cree que es mejor usar espera activa?

En general, ¿En qué condiciones cree que es mejor usar espera pasiva?

## 8. Anexos

### Anexo 1: Código fuente de apoyo, “CondCarr.c”.

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

#define REPETICIONES 20000000  /**CONSTANTE

/**VARIABLES GLOBALES (COMPARTIDAS)
    long int V = 100;          // Valor inicial

// ****FUNCIONES AUXILIARES
int test_and_set(int *spinlock) {
    int ret;
    __asm__ __volatile__(
        "xchg %0, %1"
        : "=r"(ret), "=m"(*spinlock)
        : "0"(1), "m"(*spinlock)
        : "memory");
    return ret;
}

void SeccionRestante(int V)
{
    int i;
    long tot;

    tot = 0;
    for (i = 0; i < 300; i++) {
        tot = tot+V;
    }
}

/** FUNCIONES QUE EJECUTAN LOS HILOS
void *agrega (void *argumento) {
    long int cont;
    for (cont = 0; cont < REPETICIONES; cont = cont + 1) {

        V = V + 1;

        SeccionRestante(V);
    }
    printf("-----> Fin AGREGA (V = %ld)\n", V);
    pthread_exit(0);
}

void *resta (void *argumento) {
    long int cont;
    for (cont = 0; cont < REPETICIONES; cont = cont + 1) {

        V = V - 1;

        SeccionRestante(V);
    }
    printf("-----> Fin RESTA (V = %ld)\n", V);
    pthread_exit(0);
}

void *inspecciona (void *argumento) {
    for (;;) {
        usleep(200000);
        fprintf(stderr, "Inspeccion: Valor actual de V = %ld\n", V);
    }
}

/** PROGRAMA PRINCIPAL
int main (void) {
```

```

//Declaracion de las variables necesarias.
pthread_t hiloSuma, hiloResta, hiloInspeccion;
pthread_attr_t attr;

// Inicializacion los atributos de las tareas (por defecto)
pthread_attr_init(&attr);

// EJERCICIO: Cree los tres hilos propuestos con dichos atributos
// EJERCICIO: El hilo principal debe esperar a que las
// tareas "agrega" y "resta" finalicen
// Fin del programa principal
fprintf(stderr, "-----> VALOR FINAL: V = %ld\n\n", V);
exit(0);
}

```

## Anexo 2: Sincronización por espera activa. `test_and_set`

La espera activa es una técnica de sincronización que consiste en **establecer una variable global de tipo booleano (*spinlock*) que indica si la sección crítica está ocupada**. La semántica de esta variable es: valor 0 indica FALSE y significa que la sección crítica no está ocupada; valor 1 indica TRUE y significa que la sección crítica está ocupada.

El método consiste en implementar en la sección de entrada un bucle que muestree ininterrumpidamente el valor de la variable *spinlock*. De manera que sólo pasará a ejecutar la sección crítica si está libre, pero antes de entrar deberá establecer el valor de la variable a “ocupado” (valor 1). Para hacer esto de forma segura es necesario que la operación de comprobación del valor de la variable y su asignación al valor “1” se haga de forma atómica ya que es posible que se produzca un cambio de contexto (o ejecución simultánea en computadores *multi-core*) entre la comprobación de la variable y su asignación, produciéndose así una condición de carrera en el acceso a la variable *spinlock*.

Por esta razón, los procesadores modernos incorporan en su juego de instrucciones operaciones específicas que permiten comprobar y asignar el valor a una variable de forma atómica. Concretamente, en los procesadores compatibles x86, existe una instrucción `xchg` que intercambia el valor de dos variables. Como la operación consiste en una sola instrucción máquina, su atomicidad está asegurada. Usando la instrucción `xchg`, se puede construir una función `test_and_set` que realice de forma atómica las operaciones de comprobación y asignación comentadas anteriormente. El código que implementa esta operación `test_and_set` es el que aparece en la Figura 6 y está incluido en el código de apoyo que se proporciona con esta práctica.

```

int test_and_set(int *spinlock) {
    int ret;
    __asm__ __volatile__(
        "xchg %0, %1"
        : "=r"(ret), "=m"(*spinlock)
        : "0"(1), "m"(*spinlock)
        : "memory");
    return ret;
}

```

Figura 6. Código de instrucción `test_and_set` del procesador de Intel

Aunque la comprensión del código suministrado para la función `test_and_set` no es el objetivo de esta práctica, es interesante observar cómo en lenguaje C se puede incluir código escrito en ensamblador.

Con todo esto, para asegurar la exclusión mutua en el acceso a la sección crítica utilizando este método, hay que modificar el código tal y como se indica en la Tabla 3.

<pre>//Declarar una variable <u>global</u>, el “spinlock” que usarán todos los hilos int llave = 0;    // inicialmente FALSE → sección crítica NO está ocupada.</pre>	
<b>Sección de entrada</b>	<code>while(test_and_set(&amp;llave));</code>
<b>Sección de salida</b>	<code>llave=0;</code>

Tabla 3: Protocolo de entrada y salida con `test_and_set`.

En el caso de procesadores ARM (por ejemplo, los que usan los nuevos ordenadores de Apple), no existe una instrucción idéntica a `xchg` de x86 que intercambie valores de manera atómica en una sola operación. Pero se pueden utilizar los spinlocks de gcc, que se implementan en las funciones `__sync_lock_test_and_set` y `__sync_lock_release`. A diferencia de la implementación anterior para el x86, en el protocolo de salida NO se puede hacer simplemente `llave=0`, y hay que hacer uso de la función `release` (ver código en la figura 7). En la tabla 4, se puede ver cómo queda el protocolo de entrada y salida. Estas nuevas funciones son genéricas y se pueden usar también en procesadores x86.

```
int test_and_set(int *spinlock) {
    return (__sync_lock_test_and_set(spinlock, 1));
}

void release(int *spinlock) {
    __sync_lock_release(spinlock);
}
```

Figura 7. Código de instrucción `test_and_set` y reléase usando spinlocks

<pre>//Declarar una variable <u>global</u>, el “spinlock” que usarán todos los hilos int llave = 0;    // inicialmente FALSE → sección crítica NO está ocupada.</pre>	
<b>Sección de entrada</b>	<code>while(test_and_set(&amp;llave));</code>
<b>Sección de salida</b>	<code>release(llave);</code>

Tabla 4: Protocolo de entrada y salida con `test_and_set` y `release`

### Anexo 3: Sincronización por espera pasiva. Semáforos

La espera pasiva se consigue con la ayuda del Sistema Operativo. Cuando un hilo tiene que esperar para entrar en la sección crítica (porque otro hilo está ejecutando su sección crítica), se “suspende” eliminándolo de la lista de hilos en estado “preparado” del planificador. De esta forma los hilos en espera no consumen tiempo de CPU, en lugar de esperar en un bucle de consulta como en el caso de espera activa.

Para que los programadores puedan utilizar la espera pasiva, el Sistema Operativo ofrece unos objetos específicos que se llaman “semáforos” (tipo `sem_t`). Un semáforo, ilustrado en la Figura 6, está compuesto por un contador,

cuyo valor inicial se puede fijar en el momento de su creación, y una cola de hilos suspendidos a la espera de ser reactivados. Inicialmente el contador debe ser mayor o igual a cero y la cola de procesos suspendidos está vacía. El semáforo soporta dos operaciones:

- Operación `sem_wait()` (operación P en la notación de Dijkstra): Esta operación decrementa el contador del semáforo y, si después de efectuar el decremento el contador es estrictamente menor que cero, suspende en la cola del semáforo al hilo que invocó la operación.
- Operación `sem_post()` (operación V en la notación de Dijkstra): Esta operación incrementa el contador del semáforo y, si después de efectuar el incremento el contador es menor o igual que cero, despierta al primer hilo suspendido en la cola del semáforo, aplicando una ordenación FIFO.

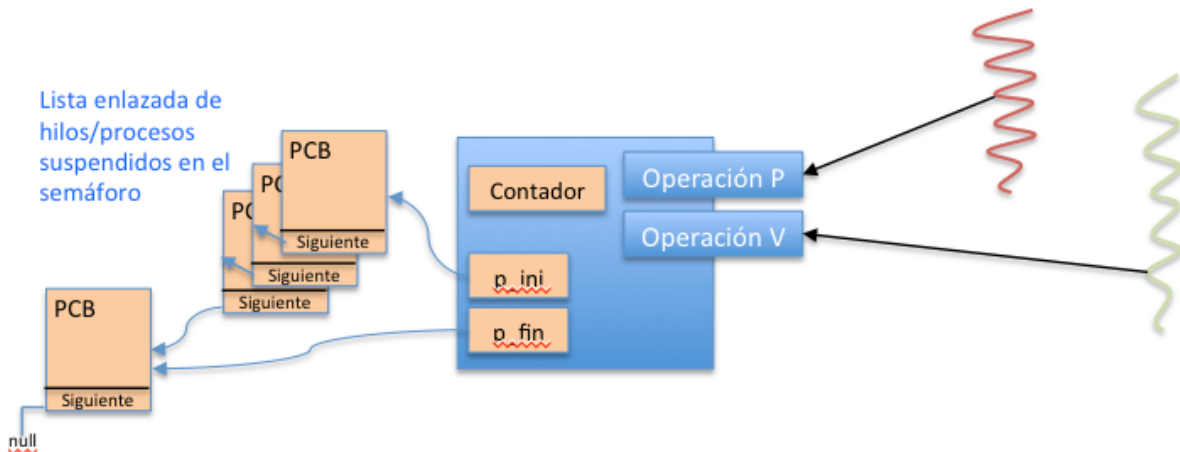


Figura 6: Estructura de un semáforo y sus operaciones.

**Nota:** Aunque los semáforos POSIX (`sem_t`) forman parte del estándar, en MacOSX no funcionan. MacOSX proporciona otros objetos (`semaphore_t`) que se comportan de manera análoga y pueden usarse para ofrecer el mismo interfaz que ofrecen los semáforos POSIX.

Dependiendo del uso que le queramos dar a un semáforo, definiremos su valor inicial. El valor inicial de un semáforo puede ser mayor o igual a cero y su semántica asociada es la de “número de recursos disponibles inicialmente”. Esencialmente un semáforo es un contador de recursos que pueden ser solicitados (con la operación `sem_wait`) y liberados (con la operación `sem_post`) de forma que cuando no hay recursos disponibles, los hilos que solicitan recursos se suspenden a la espera de que algún recurso sea liberado.

Especialmente relevantes son los semáforos con valor inicial igual a uno. Como sólo hay un recurso libre inicialmente, sólo un hilo podrá ejecutar la sección crítica en exclusión mutua con los demás. Estos semáforos se suelen llamar “mutex” y son los que nos interesan en esta práctica.

Con todo esto, para asegurar la exclusión mutua en el acceso a la sección crítica utilizando este método, hay que modificar el código tal y como se indica en la tabla 4.

<pre>//Incluir las cabeceras de la librería de semáforos. #include &lt;semaphore.h&gt; //Declarar una variable <u>global</u>, el “semáforo” que usarán todos los hilos sem_t sem; // No está inicializado, sólo declarado.</pre>	
<b>Sección de entrada</b>	<code>sem_wait(&amp;sem);</code>
<b>Sección de salida</b>	<code>sem_post(&amp;sem);</code>



<pre>//En el programa principal "main()" hay que inicializar el semáforo. sem_init(&amp;sem,0,1); // El segundo parámetro indica que el semáforo no es compartido                 // y el último parámetro indica el valor inicial,                 // "1" en nuestro caso (exclusión mutua).</pre>	

Tabla 5. Descripción del protocolo de entrada y salida a la sección crítica con semáforos

#### Anexo 4: Sincronización por espera pasiva. "Mutex de pthread"

Además de semáforos que ofrece el S.O. bajo el estándar POSIX, la librería de soporte para hilos de ejecución *pthread* proporciona otros objetos de sincronización: los *mutex* y las variables condición ("condition"). Los *mutex*, objetos `pthread_mutex_t`, se usan para resolver el problema de la exclusión mutua como indica su nombre y pueden considerarse como semáforos con valor inicial "1" y cuyo valor máximo es también "1". Obviamente son objetos específicos creados para asegurar la exclusión mutua y no pueden ser utilizados como contadores de recursos.

Al igual que se ha hecho con los otros métodos de sincronización, se muestra el uso de los "mutex de pthread" en la siguiente tabla

<pre>//Incluir las cabeceras de la librería de pthreads. Normalmente ya está incluida porque estamos usando hilos. #include &lt;pthread.h&gt; //Declarar una variable <u>global</u>, el "mutex" que usarán todos los hilos pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; //Esto lo declara e inicializa.</pre>	
<b>Sección de entrada</b>	<code>pthread_mutex_lock(&amp;mutex);</code>
<b>Sección de salida</b>	<code>pthread_mutex_unlock(&amp;mutex);</code>

Tabla 6 : Descripción del protocolo de entrada y salida a la sección crítica con Mutex.