Práctica de fso 5 resulta del año 2021

Fundamentos de sistemas operativos (Universitat Politecnica de Valencia)

PL05-Castellano-1.pdf

ainoapal

Fundamentos de sistemas operativos

2º Grado en Ingeniería Informática

Escuela Técnica Superior de Ingeniería Informática
Universitat Politècnica de València

Fundamentos de los Sistemas Operativos

Departamento de Informática de Sistemas y Computadoras (DISCA)
Universitat Politècnica de València

fSO

Práctica 5

Creación y Sincronización de Hilos POSIX (v1.0)

Contenido

1. Objetivos 3
2. Creación de hilos 3
Ejercicio 1: Trabajando con pthread_join y pthread_exit 5
3. Variables compartidas entre hilos 5
4. Observando condiciones de carrera 7
Ejercicio2: Creación de hilos “CondCarr.c” 7
Ejercicio3: Provocando condiciones de carrera 7
5. Soluciones para evitar la condición de carrera 8
6. Protegiendo sección crítica 9
Ejercicio4: Solución de sincronización con “test_and_set” 9
Ejercicio5: Solución de sincronización con semáforos 10
Ejercicio6: Solución de sincronización con mutex 10
7. Actividades Opcionales 11
8. Anexos 13
Anexo 1: Código fuente de apoyo, “CondCarr.c”. 13
Anexo 2: Sincronización por espera activa. Test_and_set 14
Anexo 3: Sincronización por espera pasiva. Semáforos 15
Anexo 4: Sincronización por espera pasiva. “Mutex de pthreas” 16

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.
- **T**rabajar 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>

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);
 }
```

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 siguientes modificaciones 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?

> No se imprime nada, porque el hilo principal no espera a thread1 y thread2, destruyéndolos al finalizar antes.

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?

> **Sí, porque pthread_exit asegura que si el hilo principal termina antes que el resto de hilos estos no serán eliminados hasta que finalice su ejecución.**

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 (us**ando la función usleep(...) de la librería <unistd.d> 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?

> A pesar de que no haya ningún join o exit, el programa se ejecuta con normalidad porque el hilo principal finaliza después que thread1 y thread2 gracias al retraso que añade usleep.

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

- ¿Qué ocurre ahora? ¿Por qué?

> No se imprime nada. Como el retraso añadido en los hilos es mayor que el añadido en el main, el main finaliza antes destruyendo los hilos antes de que terminen su ejecución.

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.

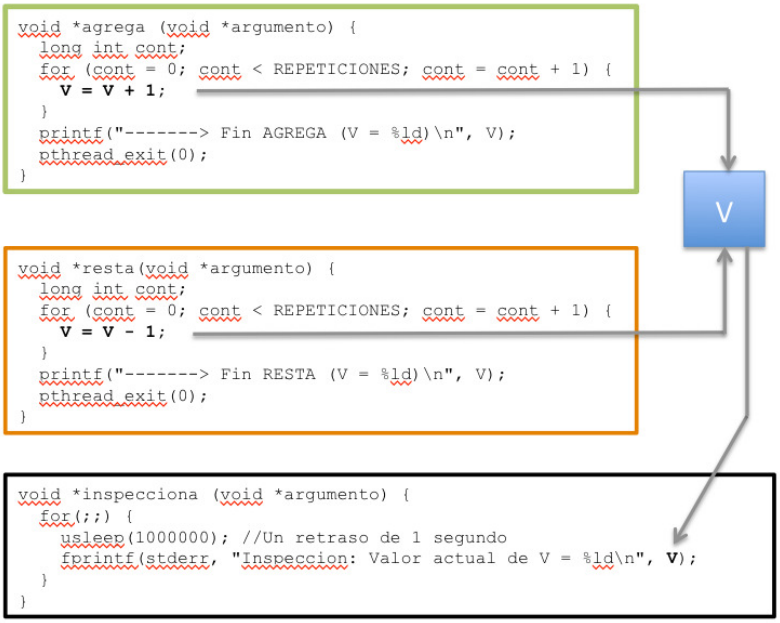

Figura-2: Código de las funciones agrega, resta e inspecciona

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. 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 y modificar las secciones de incremento y decremento, con una variable auxiliar durante las operaciones, para aumentar la probabilidad de que se produzca un cambio de contexto en medio de la operación.

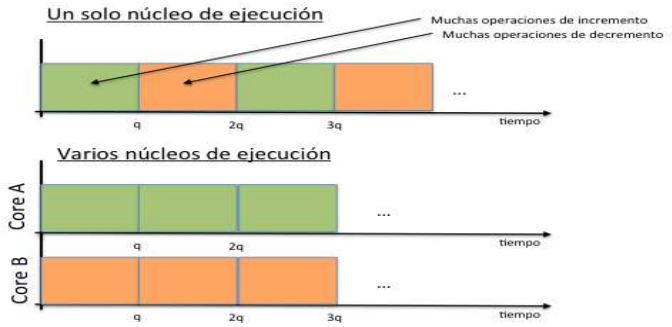

Figura-3: Ejecución de los hilos agrega y resta en CPU, en máquinas con uno y dos núcleos.

Cuando sea necesario aumentar el tiempo de cómputo de las secciones incremento y decremento, introduzca los cambios de la tabla 1. Debe declarar la variable local "aux" en cada hilo del tipo "long int".

	Código original	Sustituir por…..

agrega()	V=V+1;	aux=V;
		aux=aux+1;
		V=aux;

resta()	V=V-1	aux=V;
		aux=aux-1;
		V=aux;

Tabla 1. Escenario de incremento y decremento con variable auxiliar.

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()*, ver 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.

Valor final V = 100, por lo que no se ha producido condición de carrera. Esto se debe a lo que se explica aquí debajo:

En principio cabría esperar que el acceso concurrente a la variable **V** sin ningún tipo de protección provocara una condición de carrera de forma que el valor final de **V** fuese distinto del inicial (100). Sin embargo, para valores bajos de REPETICONES, esto puede no ocurrir observándose que el valor final de V es el inicial (100). Esto es debido a que, en sistemas con un solo procesador, no da tiempo a que los dos hilos se ejecuten concurrentemente y haya cambios de contexto. Si se crea el primer hilo, este empieza a ejecutarse, y termina antes de que empiece a ejecutarse el segundo hilo, ambos hilos no llegan a ejecutar concurrentemente. En sistemas *multi-core* es más fácil observar una condición de carrera ya que la concurrencia es real.

Ejercicio3: Provocando condiciones de carrera

Modifique el código de CondCarr.c aumentando progresivamente los valores a la constante REPETICIONES para observar ambas situaciones, es decir, la situación en la que no se observa una condición de carrera y la situación en la que si se observa. Anote para ambos casos los valores de REPETICIONES en la siguiente tabla.

REPETICIONES No se observa condición de carrera	REPETICIONES Sí se observa condición de carrera
20000000	A partir de 50000000 (aprox)

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

```
$ time ./CondCarr
```

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

> **Nota:** Si estamos trabajando con una máquina virual con un único procesador, debemos tener en cuenta que los los tiempos que obtendremos con la orden *time* serán coherentes con esta circunstancia. Si nuestros procesos trabajan sobre un único procesador, aunque se generen diferentes hilos, no tendremos la concurrencia que nos podrían ofrecer dichos hilos trabajando sobre varios *cores*. El planificador de la CPU irá asignando el uso de la CPU a los diferentes hilos, pero no habrá concurrencia en la ejecución de los mismos. Y, por tanto, los tiempos de CPU medidos por el planificador serán muy similares al tiempo real de ejecución. No habrá una mejora significativa en el tiempo real de ejecución por el hecho de utilizar hilos. En cualquier caso, será posible configurar la máquina virtual con más de un procesador, pero sólo si nuestra máquina tiene recursos suficientes.

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.c Sin proteger la sección crítica	
Tiempo real de ejecución	0m 57,621s
Tiempo de ejecución en modo usuario	0m 54,331s
Tiempo de ejecución en modo sistema	0m 0,016s

> **Nota:** Si el tiempo de ejecución es muy corto, aumente generosamente el valor REPETICIONES hasta que el tiempo real de ejecución del programa sea observable por un humano (del orden de 200ms). Esto nos proporcionará una versión del programa muy propicia para que se produzcan condiciones de carrera.

5. 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, aux;

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

Figura-4: Protocolo de entrada y protocolo de salida a la sección critica 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 "pthreads".

¡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.

6. Protegiendo sección crítica

Trabaje únicamente sobre la versión del código en la que SÍ se observan condiciones de carrera (CondCarr.c). Si el valor original de REPETICIONES ya produce condiciones de carrera utilizar éste.
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.

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 test_and_set	
Tiempo real de ejecución	0m. 2,565s
Tiempo de ejecución en modo usuario	0m. 2,339s
Tiempo de ejecución en modo sistema	0m. 0,004s

3. 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;
 long int aux;

        Protocolo de Entrada
     for (cont = 0; cont < REPETICIONES; cont = cont + 1) {
      V = V + 1;
     }
        Protocolo de  Salida
     printf("-------> Fin AGREGA (V = %ld)\n", V);
     pthread_exit(0);
}
```

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

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

CondCarrTB.c Protegiendo todo el bucle "for" con test_and_set	
Tiempo real de ejecución	0m 0,448s
Tiempo de ejecución en modo usuario	0m 0,385s
Tiempo de ejecución en modo sistema	0m 0,000s

5. Observando los resultados obtenidos identifique qué diferencia hay entre sincronizar las secciones críticas como se indica en la Figura 4 o hacerlo como indica la Figura 5.

¿Qué ha ocurrido al utilizar el esquema de sincronización de la Figura 5?

Han disminuido los tiempos de ejecución

¿Qué ventaja tiene sincronizar las secciones críticas como se indica en la Figura 4?

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 4 (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.c Protegiendo la sección crítica con semáforos sem_t	
Tiempo real de ejecución	0m 3,972s
Tiempo de ejecución en modo usuario	0m 3,564s
Tiempo de ejecución en modo sistema	0m 0,080s

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 5 (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	0m 3,423s
Tiempo de ejecución en modo usuario	0m 3,118s
Tiempo de ejecución en modo sistema	0m 0,004s

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

En esta práctica es más eficiente la espera activa.

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

La espera activa puede ser recomendable con existencia de pocos hilos y una sección crítica breve.

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

En todas las que no sean las mencionadas en el apartado anterior.

7. Actividades Opcionales

Para comprobar qué ocurre cuando la sección crítica es grande y cómo influye esto en el método de sincronización escogido, podemos aumentar la duración de la sección crítica artificialmente de forma análoga a como se propuso en la Tabla 1, pero introduciendo un retraso antes de asignar el nuevo valor a la variable compartida V. Esta es la modificación que se propone en la Tabla 2.

	Código original	Sustituir por…..

agrega()	V=V+1;	aux=V;
		aux=aux+1;
		usleep(500);
		V=aux;

resta()	V=V-1	aux=V;
		aux=aux-1;
		usleep(500);
		V=aux;

Tabla 2: Nueva sección crítica a trabajar

El pequeño retraso introducido de medio milisegundo en el código propuesto en la Tabla 2 hace que aumente considerablemente la probabilidad de que se produzca una condición de carrera además de aumentar, también considerablemente, el tiempo de ejecución del programa.

Para que el tiempo de ejecución del programa sea fácilmente observable en todos los casos, hay que disminuir el valor de la constante REPETICIONES.

1. Modifique el código de CondCarr.c, CondCarrT.c, CondCarrS.c y CondCarrM.c como indica la Tabla 2. Disminuya también en los cuatro archivos el valor de REPETICIONES para que el tiempo real de ejecución de la versión sin sincronización esté en torno al medio segundo (un valor de REPETICIONES entre 1000 y 10000 suele ser adecuado, hacer pruebas con 10000). Para que los resultados sean comparables, obviamente, debe usar el mismo valor de REPETICIONES en los cuatro archivos.

2. Ejecute con la orden ***time*** las cuatro versiones del código y anote los tiempos de ejecución.

Sección crítica larga	Sin proteger CondCarr.c	testandSet CondCarrT.c	Semaphore CondCarrS.c	Mutex CondCarrM.c
Tiempo real de ejecución				
Tiempo de ejecución en modo usuario				
Tiempo de ejecución en modo sistema				

3. Según estos resultados revise las respuestas que ha dado a las cuestiones formuladas en el ejercicio 6.

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;
 }

 //** FUNCIONES QUE EJECUTAN LOS HILOS
 void *agrega (void *argumento) {
   long int cont, aux;
       for (cont = 0; cont < REPETICIONES; cont = cont + 1) {
        V = V + 1;
       }
       printf("-------> Fin AGREGA (V = %ld)\n", V);
       pthread_exit(0);
 }
 void *resta (void *argumento) {
   long int cont,aux;
       for (cont = 0; cont < REPETICIONES; cont = cont + 1) {
        V = V - 1;
       }
       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*.

para esto

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.

//Declarar una variable global, el "spinlock" que usarán todos los hilos `int llave = 0;` `// inicialmente FALSE → sección crítica NO está ocupada.`	
Sección de entrada	`while(test_and_set(&llave));`
Sección de salida	`llave=0;`

Tabla 3: Protocolo de entrada y salida con Test_and_Set.

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.

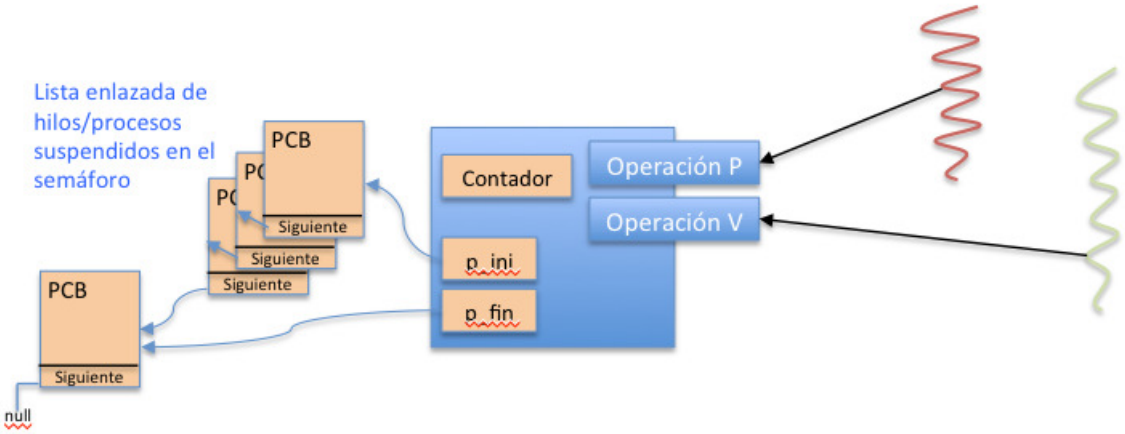

Figura 6: Estructura de un semáforo y sus operaciones.

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.

//Incluir las cabeceras de la librería de semáforos.
`#include <semaphore.h>`
//Declarar una variable global, el "semáforo" que usarán todos los hilos
`sem_t sem;  // No está inicializado, sólo declarado.`	

Sección de entrada	`sem_wait(&sem);`
Sección de salida	`sem_post(&sem);`

//En el programa principal "main()" hay que inicializar el semáforo.
`sem_init(&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).`	

Tabla 4. Descripción del protocolo de entrada y salida a la sección critica con semáforos

Anexo 4: Sincronización por espera pasiva. "Mutex de pthreas"

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 pthreads" en la siguiente tabla

//Incluir las cabeceras de la librería de pthreads. Normalmente ya está incluida porque estamos usando hilos.
`#include <pthread.h>`
//Declarar una variable global, el "mutex" que usarán todos los hilos
`pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; //Esto lo declara e inicializa.`	

Sección de entrada	`pthread_mutex_lock(&mutex);`
Sección de salida	`pthread_mutex_unlock(&mutex);`

Tabla 5 : Descripción del protocolo de entrada y salida a la sección critica con Mutex.