TEC | Tecnológico de Costa Rica

Escuela de Ingeniería en Computación
411 Ingeniería en Computación - 2018
IC6600 - Principios de Sistemas Operativos

Tarea 2

Biblioteca Sync

Ávila Ramírez Paublo Alexander
pavila@estudiantec.cr
2022035584

Reyes Rodríguez Ricardo Andrés
rireyes@estudiantec.cr
2022101681

Zúñiga Campos Omar Jesús
omzuniga@estudiantec.cr
2022019053

San José, Costa Rica
Agosto 2024

Índice general

Capítulo 1

Introducción

Cuando se habla de programación concurrente y paralela, un hilo corresponde a una unidad de ejecución la cual dentro de un proceso es capaz de ejecutarse de una forma independiente. Los hilos son muy estudiados y aplicados debido a que permiten llevar a cabo múltiples tareas de una forma concurrente dentro de un sistema, de tal forma que se aprovecha adecuadamente de la capacidad de los procesadores actuales para poder manejar múltiples tareas de forma simultánea. El uso de los hilos se ha vuelto básicamente una parte fundamental dentro de aplicaciones las cuales necesitan llevar a cabo operaciones paralelas o en segundo plano, como es el caso de los sistemas operativos, aplicaciones de procesamiento en tiempo real o hasta servidores web, todo esto gracias a la naturaleza de los hilos (Casero, 2024).

La sincronización de procesos es un concepto fundamental a la hora de abarcar un contexto de programación multihilo, debido a que se asegura que los hilos puedan acceder a recursos compartidos de forma segura y ordenada para que así se evitan problemas como es el caso de condiciones de carrera, donde el resultado de ejecución depende del orden en que se ejecutan los hilos y esto puede llevar a resultados que no son consistentes o hasta fallos dentro del sistema. Por lo tanto, la sincronización es vital para coordinar las acciones de los hilos y que así haya un seguro de que estos sigan reglas de acceso adecuadas a recursos compartidos, que pueden ser: Memoria, archivos o en general, cualquier tipo de datos que se esté compartiendo (Casero, 2024).

En el caso del lenguaje de programación C, los hilos se gestionan de forma principal por medio de la biblioteca pthread que es una abreviación de POSIX threads, la cual brinda un conjunto de funciones que facilitan la creación y manejo de hilos así como la implementación de mecanismos de sincronización como las que se pueden apreciar en este trabajo en el capítulo 2. Por medio de pthread, los desarrolladores son capaces de crear aplicaciones que emplean múltiples hilos con el fin de ver una mejora en la eficiencia y rendimiento, sobre todo en sistemas multiprocesador, que ahí es donde se aprovecha la capacidad

para poder ejecutar varios hilos en paralelo (Casero, 2024).

Como tal, en este documento se abordan diferentes métodos para la sincronización de procesos como son: Mutex, semáforo, barrera y bloqueos de lectura/escritura, que son empleados en la gestión para el acceso de manera concurrente a recursos compartidos. Cada uno de estos métodos son abordados en más detalle en el capítulo 2, donde se abarca: Qué son, características, usos, ventajas, desventajas, implementación y ejemplo, donde estos últimos dos también se pueden encontrar en el repositorio de github: `https://github.com/ch0so/biblioteca_sync`, todo esto con el fin de brindar así un informe detallado de estas herramientas empleadas para la resolución de problemas de sincronización que pueden llegar a ocurrir en una aplicación multihilo.

1.1. Antecedentes

El área de la ingeniería en computación siempre está en constante evolución, sobre todo porque es una disciplina que abarca una gama bastante amplía de temas y entre estos, están los sistemas informáticos que fueron evolucionado hacía una arquitectura de multiprocesador, impulsando así la necesidad de que se desarrollen programas capaces de sacar provecho a los múltiples núcleos de procesamiento. Por lo tanto, el uso de hilos y la sincronización de procesos se ha vuelto una parte vital para poder mejorar el rendimiento y eficiencia de aplicaciones (GeeksforGeeks, 2023).

En un inicio, los programas informáticos eran ejecutados de forma secuencial, por lo que una instrucción tenía que completarse antes de que la siguiente pudiera dar inicio. Sin embargo, al aparecer los sistemas que contaban con unidades de procesamiento central (CPU), emergió la posibilidad de ejecutar múltiples instrucciones de manera simultánea. Los hilos permitieron que diversas partes de un programa fueran ejecutada en paralelo, permitiendo un mejor tiempo de respuesta y uso del CPU (Casero, 2024).

EL uso de hilos, pese a que aprovechaba de gran manera la nueva arquitectura, también introdujo más desafíos debido a que al acceder múltiples hilos a los mismos recursos compartidos, es posible que aparezcan problemas como es el caso de las condiciones de carrera, donde dos o más hilos están intentando modificar un recurso al mismo tiempo, siendo así que terminan por ocasionar resultados no deseados y errores difíciles de encontrar. Estos problemas hicieron evidente la necesidad de crear mecanismos de sincronización eficaces que permitan una adecuada coordinación de los hilos a la hora de acceder a los recursos compartidos (Casero, 2024).

Una forma de atacar los desafíos mencionados en el párrafo anterior, fue a través de estructuras de sincronización, como es el caso de mutex (Exclusión

mutua), semáforos, barreras y bloqueos de lectura/escritura. Cada una de estas estructuras o mecanismos brinda diversas maneras de controlar el acceso que se da de forma concurrente a los recursos, siendo así que asegura que los hilos no interfieran los unos con los otros destructivamente. La biblioteca pthread, que es un acrónimo de POSIX threads, terminó por convertir en el estándar para la programación multihilo en el lenguaje de programación C, ya que brinda un conjunto de herramientas que permiten crear y gestionar hilos, así como implementar cada una de estas estructuras (Casero, 2024).

Con el paso del tiempo, ha ido aumentando la necesidad de crear aplicaciones cada vez más rápidas y eficientes, por lo que el enfoque ha sido la programación concurrente y una optimización del uso de recursos por medio de hilos y sincronización. El desarrollo de todo esto ha sido vital para poder crear sistemas operativos modernos, aplicaciones de alto rendimiento y servicios que necesitan del manejo de múltiples tareas de forma simultánea con una elevada fiabilidad y eficiente (Casero, 2024).

Capítulo 2

Estructuras de sincronización

2.1. Mutex

2.1.1. Explicación del concepto

Un **mutex**, que es una abreviatura de *mutual exclusion* o exclusión mutua en español, es un mecanismo empleado en la programación concurrente para asegurar que múltiples procesos o hilos no accedan de manera simultánea a un recurso compartido, siendo este el caso de una variable o región crítica de código, lo cual se consigue al bloquear el acceso al recurso al ser utilizado por un proceso, siendo así que otros procesos no pueden acceder a ese recurso hasta que sea liberado(IBM, 2023a).

Generalmente, mutex es utilizado en sistemas operativos y aplicaciones de software que necesitan de un acceso sincronizado a recursos compartidos, como puede ser la protección de datos compartidos, donde múltiples hilos o procesos están accediendo y modificando datos que son compartidos, por lo que un mutex permite que solo uno pueda hacerlo a la vez para que no se corrompan los datos. Otro ejemplo más claro es cuando se emplean en la coordinación de la ejecución de hilos dentro de una aplicación multihilo para que secciones específicas de código sean ejecutadas de forma ordenada y que así no presenten interferencias. Como último ejemplo, se presentan las secciones de código que no se interrumpa para que a través de los mutex sean ejecutados correctamente (Fernando, 2011).

Los ejemplos anteriores dejan en claro que mutex ofrece una serie de ventajas a la hora de la programación concurrente, entre las cuales está el hecho de que evita condiciones de carrera debido a que garantiza que un único hilo o proceso acceda a un recurso compartido a la vez, siendo así que se eliminan las condiciones de carrera en las cuales el resultado del programa depende del orden en el

cual se ejecutan los hilos. A su vez, esto permite que haya una mejor integridad de datos ya que los datos compartidos no se modifican al mismo tiempo por múltiples hilos, siendo así que brinda una mejor consistencia e integridad a este tipo de datos. Además, permite una sincronización efectiva entre distintos hilos, siendo así que permite una adecuada coordinación para ejecutar tareas que son dependientes las unas con las otras (Fernando, 2011).

Si bien esta estructura de sincronización presenta múltiples ventajas, también tiene una serie de desventajas, como es el caso de posibles bloqueos en el caso que no se gestionan bien, los cuales se conocen comúnmente como deadlocks, en los cuales dos o más hilos esperan de forma indefinida por recursos que nunca van a ser liberados. Otra desventaja es el hecho que puede haber un rendimiento reducido por la sobrecarga de bloqueo y desbloqueo, sobre todo en aplicaciones que emplean bastantes hilos o procesos. A esto se le suma el hecho de que pueden añadir complejidad adicional ya que manejar un mutex correctamente agrega una mayor complejidad en el diseño e implementación del software, por lo que va a requerir de un esfuerzo mayor tanto en el apartado de programación como en el depuración, que es muy importante a considerar en proyectos con recursos limitados (Fernando, 2011).

Básicamente, esta estructura de sincronización es una herramienta fundamental en el área de programación concurrente, debido a que permite garantizar una exclusión mutua al mismo tiempo que brinda una mayor integridad de los datos pero necesitan que se maneje con sumo cuidado para que no hayan problemas como deadlocks o reducción de rendimiento por un desconocimiento por parte de programador o programadora (IBM, 2023a).

2.1.2. Descripción de la implementación

Estructura del mutex

```
1  typedef struct {
2      pthread_mutex_t mutex;
3  } mutex_t;
```

- La estructura `mutex_t` se utiliza para encapsular el mutex de la biblioteca de pthreads.

Función mutex_init

```
1  void mutex_init(mutex_t *parameter_mutex) {
2      pthread_mutex_init(&parameter_mutex->mutex, NULL);
3  }
```

- **Descripción:**

- La función `mutex_init` se encarga de inicializar un mutex.

- **Entradas:**
  - `parameter_mutex`: Un puntero a la estructura del mutex que contiene el mutex a inicializar.
- **Salidas:**
  - Ninguna.
- **Restricciones:**
  - La variable `parameter_mutex` no debe ser `NULL` y debe apuntar a un área de memoria válida.

Función mutex_lock

```
1 void mutex_lock(mutex_t *parameter_mutex) {
2     pthread_mutex_lock(&parameter_mutex->mutex);
3 }
```

- **Descripción:**
  - La función `mutex_lock` bloquea un mutex para sincronización entre hilos.
- **Entradas:**
  - `parameter_mutex`: Un puntero a la estructura del mutex que contiene el mutex a bloquear.
- **Salidas:**
  - Ninguna.
- **Restricciones:**
  - La variable `parameter_mutex` no debe ser `NULL` y debe apuntar a un área de memoria válida.
  - Se asume que el mutex ha sido previamente inicializado con `mutex_init`.

Función mutex_unlock

```
1 void mutex_unlock(mutex_t *parameter_mutex) {
2     pthread_mutex_unlock(&parameter_mutex->mutex);
3 }
```

- **Descripción:**

- La función `mutex_unlock` desbloquea un mutex, permitiendo que otros hilos continúen su ejecución.

- **Entradas:**
  - `parameter_mutex`: Un puntero a la estructura del mutex que contiene el mutex a desbloquear.
- **Salidas:**
  - Ninguna.
- **Restricciones:**
  - La variable `parameter_mutex` no debe ser `NULL` y debe apuntar a un área de memoria válida.
  - Se asume que el mutex ha sido previamente inicializado con `mutex_init` y bloqueado por el hilo actual.

Función mutex_destroy

```
1  void mutex_destroy(mutex_t *parameter_mutex) {
2      pthread_mutex_destroy(&parameter_mutex->mutex);
3  }
```

- **Descripción:**
  - La función `mutex_destroy` destruye un mutex, liberando los recursos asociados.
- **Entradas:**
  - `parameter_mutex`: Un puntero a la estructura del mutex que contiene el mutex a destruir.
- **Salidas:**
  - Ninguna.
- **Restricciones:**
  - La variable `parameter_mutex` no debe ser `NULL` y debe apuntar a un área de memoria válida.
  - Se asume que el mutex ha sido previamente inicializado con `mutex_init`.

2.1.3. Descripción del ejemplo

- El ejemplo simula a una persona que espera, usa y luego deja una silla de espera, utilizando un mutex para asegurar acceso exclusivo a la silla.
- Es importante aclarar que el ejemplo forma parte del main.c del repositorio de GitHub, por lo que aquí se añade únicamente lo referente al ejemplo del mutex pero en el main.c está incluido dentro del menú de opciones.
- Inclusión de bibliotecas necesarias:

```
1    #include <stdio.h>
2    #include <stdlib.h>
3    #include <pthread.h>
4    #include <unistd.h>
```

  Estas líneas de código son empleadas para imporntar las bibliotecas estándar de C necesarias para manejar entradas y salidas (stdio.h), asignación de memoria (stdlib.h), creación y manejo de hilos (pthread.h), y funciones relacionadas con el tiempo (unistd.h).

- Definición de constantes y variables:

```
1    #define NUM_PERSONS 5
2    mutex_t mutex;
3    pthread_t threads[NUM_PERSONS];
```

  Aquí se define una constante NUM_PERSONS que establece el número de personas que simularán usando la silla, en este caso, 5. También se declara una variable de tipo mutex_t llamada mutex para manejar la sincronización, y un arreglo threads de tipo pthread_t que almacenará los identificadores de los hilos creados.

- Inicialización del mutex:

```
1    mutex_init(&mutex);
```

  Se inicializa el mutex utilizando la función mutex_init, asegurando que el mutex esté listo para usarse antes de la creación de los hilos.

- Creación de hilos:

```
1    for (int i = 0; i < NUM_PERSONS; i++) {
2        int* person_id = malloc(sizeof(int));
3        *person_id = i;
4        pthread_create(&threads[i], NULL, mutex_example,
             person_id);
5    }
```

  En este bloque, se crea un bucle que itera NUM_PERSONS veces para crear los hilos. En cada iteración:

  - Se asigna memoria dinámica para un entero que representará el identificador de la persona person_id).

- Se asigna a *person_id el valor del índice i, que actúa como identificador único para cada persona.
- Se llama a pthread_create para crear un hilo que ejecutará la función mutex_example, pasándole como argumento el puntero person_id.

- Espera a que terminen los hilos:

```
1       for (int i = 0; i < NUM_PERSONS; i++) {
2           pthread_join(threads[i], NULL);
3       }
```

  Este bloque utiliza otro bucle para esperar a que todos los hilos terminen su ejecución. pthread_join es una función de bloqueo que asegura que el hilo principal espere la terminación de cada hilo hijo antes de continuar, garantizando que todas las personas hayan usado y dejado la silla antes de que el programa continúe.

- Destrucción del mutex:

```
1       mutex_destroy(&mutex);
```

  Después de que todos los hilos han terminado su ejecución, el mutex se destruye usando la función mutex_destroy. Esto libera cualquier recurso del sistema asociado con el mutex, asegurando una limpieza adecuada.

- Finalización del programa principal:

```
1       return 0;
```

  Finalmente, la función main retorna 0, indicando que el programa ha finalizado correctamente.

- A continuación, se encuentra todo el código sin seccionar:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4  #include <unistd.h>
5
6  #define NUM_PERSONS 5
7
8  typedef struct {
9      pthread_mutex_t mutex;
10 } mutex_t;
11
12 void mutex_init(mutex_t *parameter_mutex) {
13     pthread_mutex_init(&parameter_mutex->mutex, NULL);
14 }
15
16 void mutex_lock(mutex_t *parameter_mutex) {
17     pthread_mutex_lock(&parameter_mutex->mutex);
18 }
19
20 void mutex_unlock(mutex_t *parameter_mutex) {
```

```
21      pthread_mutex_unlock(&parameter_mutex->mutex);
22  }
23
24  void mutex_destroy(mutex_t *parameter_mutex) {
25      pthread_mutex_destroy(&parameter_mutex->mutex);
26  }
27
28  mutex_t mutex;
29
30  void* mutex_example(void* arg) {
31      int person_id = *(int*)arg;
32
33      printf("Person %d is waiting to use the chair...\n", person_id)
            ;
34      sleep(rand() % 3);
35
36      mutex_lock(&mutex);
37      printf("Person %d is sitting on the chair.\n", person_id);
38      sleep(2);
39      printf("Person %d is leaving the chair.\n", person_id);
40      mutex_unlock(&mutex);
41
42      free(arg);
43      return NULL;
44  }
45
46  int main() {
47      pthread_t threads[NUM_PERSONS];
48
49      mutex_init(&mutex);
50
51      for (int i = 0; i < NUM_PERSONS; i++) {
52          int* person_id = malloc(sizeof(int));
53          *person_id = i;
54          pthread_create(&threads[i], NULL, mutex_example, person_id)
                ;
55      }
56
57      for (int i = 0; i < NUM_PERSONS; i++) {
58          pthread_join(threads[i], NULL);
59      }
60
61      mutex_destroy(&mutex);
62      return 0;
63  }
```

2.2. Semáforo

2.2.1. Explicación del concepto

Un **semáforo** se refiere a una variable o de forma un poco más abstracta, un concepto de programación el cual se emplea para el control de acceso a un recursos compartido por varios procesos dentro de un entorno de programación concurrente. Con diferencia de mutex que solamente deja a un solo proceso en-

trar a la sección crítica, un semáforo puede dar acceso a varios recursos hasta cierto límite. En sí, consiste en un mecanismo el cual se utiliza para la sincronización, que consiste en una variable entera que se acompaña de las operaciones de esperar (En inglés wait) y señalar (En inglés signal), donde esperar o wait va disminuyendo el valor del semáforo en caso de que sea mayor que cero y en caso de ser cero, el proceso se bloquea hasta volver a tener un valor positivo. Por otro lado, la operación de señalar aumenta el valor del semáforo y en caso de haber un proceso bloqueado, se despierta el semáforo (Brooks, 2024).

En cuanto a sus usos, los semáforos son utilizados para controlar el acceso que se le da a múltiples recursos, ya que permite que varios procesos acceden de manera simultánea a un mismo recurso compartido hasta un número máximo de veces que ya está definido, como puede ser el caso de un conjunto de conexiones que se realizan a una base de datos de manera limitada. Por otra parte, también se emplean para poder contar recursos que se encuentran disponibles dentro de un sistema concurrente, permitiendo así garantizar que no se están sobrepasando los recursos asignados. Finalmente, permite una sincronización de procesos más sencilla lo que permite que haya una mejor coordinación para ejecutar tareas que son independientes las unas de las otras (GeeksforGeeks, 2024).

Al igual que mutex, este tipo de estructura también presenta varias ventajas, como es el caso de una flexibilidad al permitir que se puedan implementar distintos patrones de sincronización como es el caso de mutex, barreras o señales. Además, permite tener un mejor control sobre recursos limitados ya que permiten el acceso limitado, lo cual es de gran utilidad cuando estos recursos son escasos. Otra ventaja es que permiten una sincronización efectiva, ya que permite una coordinación entre procesos concurrentes de forma eficiente sin que se tenga que hacer un monitoreo constante del estado que hay entre los procesos (Brooks, 2024).

Como toda herramienta, también presenta sus desventajas, como es el caso de una complejidad en su manejo y esto puede ocasionar errores como son condiciones de carrera o bloqueos al mismo tiempo que pueden resultar en ser ineficientes en caso de que no se optimice adecuadamente para el caso concreto de aplicación lo que puede llevar a un elevado riesgo de bloqueo en caso de que no se gestionen adecuadamente (GeeksforGeeks, 2024).

Como toda herramienta, también presenta sus desventajas, como es el caso de una complejidad en su manejo y esto puede ocasionar errores como son condiciones de carrera o bloqueos al mismo tiempo que pueden resultar en ser ineficientes en caso de que no se optimice adecuadamente para el caso concreto de aplicación lo que puede llevar a un elevado riesgo de bloqueo en caso de que no se gestionen adecuadamente (GeeksforGeeks, 2024).

En sí, los semáforos son herramientas poderosas y flexibles en cuanto a la sincronización de procesos se trata pero al mismo tiempo deben gestionarse con

cuidado para que no haya problemas de concurrencia. Además, en la documentación que fue brindada para este estudio, se indica que existe una implementación de semáforos binarios, la cual permite el acceso exclusivo a un recurso mientras que los semáforos de conteo permiten el acceso hasta un límite en concreto. Ambos semáforos fueron implementados en este trabajo y serán abarcados en el siguiente apartado, haciendo hincapié que deben usarse de manera efectiva para que se eviten los errores anteriormente mencionados (GeeksforGeeks, 2024).

2.2.2. Descripción de la implementación

Estructura del semáforo binario

```
1  typedef struct {
2      pthread_mutex_t mutex;
3      pthread_cond_t conditional;
4      int value;
5  } binary_semaphore_t;
```

- La estructura `binary_semaphore_t` para representar al semáforo binario, donde mutex se usa para proteger el acceso al semáforo, conditional es una variable de condición para sincronización y value un número entero que indica el estado del semáforo, donde 0 es para bloqueado y 1 para desbloqueado.

Función binary_semaphore_init

```
1      pthread_mutex_init(&binary_semaphore->mutex, NULL);
2      pthread_cond_init(&binary_semaphore->conditional, NULL);
3      binary_semaphore->value = value;
4  }
```

- Esta función inicializa un semáforo binario configurando su mutex y variable de condición.
- Entradas:
  - binary_semaphore: Puntero a la estructura que contiene un mutex y una variable de condición.
  - value: Valor inicial del semáforo que puede ser 0 o 1.
- Salidas: Ninguna.
- Restricciones: binary_semaphore no debe ser NULL y debe apuntar a un área de memoria válida, y value debe ser 0 o 1.

Función binary_semaphore_wait

```
1  void binary_semaphore_wait(binary_semaphore_t *binary_semaphore) {
2      pthread_mutex_lock(&binary_semaphore->mutex);
3      while (binary_semaphore->value == 0) {
4          pthread_cond_wait(&binary_semaphore->conditional, &
               binary_semaphore->mutex);
5      }
6      binary_semaphore->value = 0;
7      pthread_mutex_unlock(&binary_semaphore->mutex);
8  }
```

- Esta función bloquea el hilo si el valor del semáforo es 0, esperando a que se convierta en 1.
- Entradas:
  - binary_semaphore: Puntero a la estructura que contiene un mutex y una variable de condición.
- Salidas: Ninguna.
- Restricciones: binary_semaphore no debe ser NULL y debe apuntar a un área de memoria válida.

Función binary_semaphore_post

```
1  void binary_semaphore_post(binary_semaphore_t *binary_semaphore) {
2      pthread_mutex_lock(&binary_semaphore->mutex);
3      binary_semaphore->value = 1;
4      pthread_cond_signal(&binary_semaphore->conditional);
5      pthread_mutex_unlock(&binary_semaphore->mutex);
6  }
```

- Esta función cambia el valor del semáforo a 1 y despierta a un hilo en espera.
- Entradas:
  - binary_semaphore: Puntero a la estructura que contiene un mutex y una variable de condición.
- Salidas: Ninguna.
- Restricciones: binary_semaphore no debe ser NULL y debe apuntar a un área de memoria válida.

Función binary_semaphore_destroy

```
1  void binary_semaphore_destroy(binary_semaphore_t *binary_semaphore)
       {
2      pthread_mutex_destroy(&binary_semaphore->mutex);
3      pthread_cond_destroy(&binary_semaphore->conditional);
4  }
```

- Esta función destruye el semáforo binario, liberando los recursos asociados.
- Entradas:
  - binary_semaphore: Puntero a la estructura que contiene un mutex y una variable de condición.
- Salidas: Ninguna.
- Restricciones: binary_semaphore no debe ser NULL y debe apuntar a un área de memoria válida.

Estructura del semáforo de conteo

```
1  typedef struct {
2      pthread_mutex_t mutex;
3      pthread_cond_t conditional;
4      int value;
5  } counting_semaphore_t;
```

- La estructura `binary_semaphore_t` para representar al semáforo binario, donde mutex se usa para proteger el acceso al semáforo, conditional es una variable de condición para sincronización y value un número entero que indica el contador del semáforo.

Función counting_semaphore_init

```
1  void counting_semaphore_init(counting_semaphore_t *
       counting_semaphore, int value) {
2      pthread_mutex_init(&counting_semaphore->mutex, NULL);
3      pthread_cond_init(&counting_semaphore->conditional, NULL);
4      counting_semaphore->value = value;
5  }
```

- Función que inicializa un semáforo de conteo con un valor inicial.
- Entradas:
  - counting_semaphore: Puntero a la estructura del semáforo de conteo que contiene un mutex, una variable de condición, y un contador
  - value: Número entero que indica el valor inicial del semáforo; representa el número de recursos disponibles.

- Salidas: Ninguna.
- Restricciones: countng_semaphore no debe ser NULL y debe apuntar a un área de memoria válida, y value debe ser un número entero no negativo

Función counting_semaphore_wait

```
1  void counting_semaphore_wait(counting_semaphore_t *
       counting_semaphore) {
2      pthread_mutex_lock(&counting_semaphore->mutex);
3      while (counting_semaphore->value <= 0) {
4          pthread_cond_wait(&counting_semaphore->conditional, &
               counting_semaphore->mutex);
5      }
6      counting_semaphore->value--;
7      pthread_mutex_unlock(&counting_semaphore->mutex);
8  }
```

- Función que decrementa el contador del semáforo de conteo, bloqueando si no hay recursos disponibles.
- Entradas:
  - counting_semaphore: Puntero a la estructura del semáforo de conteo que contiene un mutex, una variable de condición, y un contador.
- Salidas: Ninguna.
- Restricciones: counting_semaphore no debe ser NULL y debe apuntar a un área de memoria válida. Además, se asume que el semáforo ha sido previamente inicializado con counting_semaphore_init.

Función counting_semaphore_post

```
1  void counting_semaphore_post(counting_semaphore_t *
       counting_semaphore) {
2      pthread_mutex_lock(&counting_semaphore->mutex);
3      counting_semaphore->value++;
4      pthread_cond_signal(&counting_semaphore->conditional);
5      pthread_mutex_unlock(&counting_semaphore->mutex);
6  }
```

- Función que incrementa el contador del semáforo de conteo, señalando a cualquier hilo bloqueado que puede continuar
- Entradas:
  - counting_semaphore: Puntero a la estructura del semáforo de conteo que contiene un mutex, una variable de condición, y un contador.
- Salidas: Ninguna.

- Restricciones: counting_semaphore no debe ser NULL y debe apuntar a un área de memoria válida. Además, se asume que el semáforo ha sido previamente inicializado con counting_semaphore_init.

Función counting_semaphore_destroy

```
1  void counting_semaphore_destroy(counting_semaphore_t *
       counting_semaphore) {
2      pthread_mutex_destroy(&counting_semaphore->mutex);
3      pthread_cond_destroy(&counting_semaphore->conditional);
4  }
```

- Esta función destruye el semáforo de conteo, liberando los recursos asociados.
- Entradas:
  - counting_semaphore: Puntero a la estructura del semáforo de conteo que contiene un mutex, una variable de condición, y un contador
- Salidas: Ninguna.
- Restricciones: counting_semaphore no debe ser NULL y debe apuntar a un área de memoria válida. Además, se asume que el semáforo ha sido previamente inicializado con counting_semaphore_init.

2.2.3. Descripción del ejemplo

- El ejemplo simula una situación en la que varios productores generan productos y varios consumidores consumen esos productos, utilizando semáforos binarios para coordinar el acceso al buffer compartido.
- Es importante aclarar que el ejemplo forma parte del main.c del repositorio de GitHub, por lo que aquí se añade únicamente lo referente al ejemplo del semáforo binario, pero en el main.c está incluido dentro del menú de opciones.
- Inclusión de bibliotecas necesarias:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4  #include <unistd.h>
```

  Estas líneas de código son empleadas para importar las bibliotecas estándar de C necesarias para manejar entradas y salidas (stdio.h), asignación de memoria (stdlib.h), creación y manejo de hilos (pthread.h), y funciones relacionadas con el tiempo (unistd.h).

- Definición de constantes y variables:

```
1  #define BUFFER_SIZE 5
2  #define NUM_PRODUCERS 4
3  #define NUM_CONSUMERS 4
4  #define NUM_ITEMS 20
5
6  int produced_items = 0;
7  int consumed_items = 0;
8
9  binary_semaphore_t empty;
10 binary_semaphore_t full;
11
12 pthread_t producers[NUM_PRODUCERS], consumers[NUM_CONSUMERS];
13 int ids[NUM_PRODUCERS + NUM_CONSUMERS];
```

Aquí se definen constantes para el tamaño del buffer, el número de productores y consumidores, y el número total de productos. También se declaran variables para el conteo de productos producidos y consumidos, los semáforos binarios, y los identificadores de los hilos.

- Inicialización de semáforos binarios:

```
1  binary_semaphore_init(&empty, BUFFER_SIZE);
2  binary_semaphore_init(&full, 0);
```

Se inicializan los semáforos binarios empty y full. empty se inicializa con el valor del tamaño del buffer, mientras que full se inicializa en 0, ya que inicialmente el buffer está vacío.

- Creación de hilos productores:

```
1  for (int i = 0; i < NUM_PRODUCERS; i++) {
2      ids[i] = i;
3      pthread_create(&producers[i], NULL, producer_binary, &ids[
           i]);
4  }
```

Se crea un bucle para crear hilos productores. En cada iteración:

- Se asigna el identificador del productor en el arreglo ids.
- Se llama a pthread_create para crear un hilo que ejecutará la función producer_binary, pasándole el puntero al identificador del productor.

- Creación de hilos consumidores:

```
1  for (int i = 0; i < NUM_CONSUMERS; i++) {
2      ids[i + NUM_PRODUCERS] = i;
3      pthread_create(&consumers[i], NULL, consumer_binary, &ids[
           i + NUM_PRODUCERS]);
4  }
```

Similar al bloque anterior, pero para crear hilos consumidores. Los identificadores de los consumidores se asignan a partir del índice NUM_PRODUCERS en el arreglo ids.

- Espera a que terminen los hilos productores:

```
1  for (int i = 0; i < NUM_PRODUCERS; i++) {
2      pthread_join(producers[i], NULL);
3  }
```

Este bloque espera a que todos los hilos productores terminen su ejecución usando pthread_join.

- Espera a que terminen los hilos consumidores:

```
1  for (int i = 0; i < NUM_CONSUMERS; i++) {
2      pthread_join(consumers[i], NULL);
3  }
```

Similar al bloque anterior, pero para los hilos consumidores.

- Destrucción de semáforos binarios:

```
1  binary_semaphore_destroy(&empty);
2  binary_semaphore_destroy(&full);
```

Después de que todos los hilos han terminado, se destruyen los semáforos binarios para liberar los recursos del sistema.

- Finalización del programa principal:

```
1  return 0;
```

Finalmente, la función main retorna 0, indicando que el programa ha finalizado correctamente.

- A continuación, se encuentra todo el código sin seccionar:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4  #include <unistd.h>
5
6  #define BUFFER_SIZE 5
7  #define NUM_PRODUCERS 4
8  #define NUM_CONSUMERS 4
9  #define NUM_ITEMS 20
10
11 int produced_items = 0;
12 int consumed_items = 0;
13 int in = 0;
14 int out = 0;
15 int buffer[BUFFER_SIZE];
16
17 typedef struct {
18     pthread_mutex_t mutex;
19     pthread_cond_t conditional;
20     int value;
21 } binary_semaphore_t;
22
23
```

```
24 void binary_semaphore_init(binary_semaphore_t *binary_semaphore,
       int value) {
25     pthread_mutex_init(&binary_semaphore->mutex, NULL);
26     pthread_cond_init(&binary_semaphore->conditional, NULL);
27     binary_semaphore->value = value;
28 }
29
30 void binary_semaphore_wait(binary_semaphore_t *binary_semaphore) {
31     pthread_mutex_lock(&binary_semaphore->mutex);
32     while (binary_semaphore->value == 0) {
33         pthread_cond_wait(&binary_semaphore->conditional, &
               binary_semaphore->mutex);
34     }
35     binary_semaphore->value = 0;
36     pthread_mutex_unlock(&binary_semaphore->mutex);
37 }
38
39 void binary_semaphore_post(binary_semaphore_t *binary_semaphore) {
40     pthread_mutex_lock(&binary_semaphore->mutex);
41     binary_semaphore->value = 1;
42     pthread_cond_signal(&binary_semaphore->conditional);
43     pthread_mutex_unlock(&binary_semaphore->mutex);
44 }
45
46 void binary_semaphore_destroy(binary_semaphore_t *binary_semaphore)
       {
47     pthread_mutex_destroy(&binary_semaphore->mutex);
48     pthread_cond_destroy(&binary_semaphore->conditional);
49 }
50
51 binary_semaphore_t empty;
52 binary_semaphore_t full;
53
54 pthread_t producers[NUM_PRODUCERS], consumers[NUM_CONSUMERS];
55 int ids[NUM_PRODUCERS + NUM_CONSUMERS];
56
57 void* producer_binary(void* arg) {
58     int id = *((int*) arg);
59     for (int i = 0; i < NUM_ITEMS / NUM_PRODUCERS; i++) {
60         int item = rand() % 100;
61         binary_semaphore_wait(&empty);
62         buffer[in] = item;
63         printf("Producer %d: produced %d\n", id, item);
64         in = (in + 1) % BUFFER_SIZE;
65         produced_items++;
66         binary_semaphore_post(&full);
67
68         if (produced_items >= NUM_ITEMS) {
69             break;
70         }
71         sleep(1);
72     }
73     return NULL;
74 }
75
76 void* consumer_binary(void* arg) {
77     int id = *((int*) arg);
```

```
78      while (consumed_items < NUM_ITEMS) {
79          binary_semaphore_wait(&full);
80          int item = buffer[out];
81          printf("Consumer %d: consumed %d\n", id, item);
82          out = (out + 1) % BUFFER_SIZE;
83          consumed_items++;
84          binary_semaphore_post(&empty);
85
86          if (consumed_items >= NUM_ITEMS) {
87              break;
88          }
89          sleep(1);
90      }
91      return NULL;
92  }
93
94  int main() {
95      binary_semaphore_init(&empty, BUFFER_SIZE);
96      binary_semaphore_init(&full, 0);
97
98      for (int i = 0; i < NUM_PRODUCERS; i++) {
99          ids[i] = i;
100         pthread_create(&producers[i], NULL, producer_binary, &ids[i
                ]);
101     }
102
103     for (int i = 0; i < NUM_CONSUMERS; i++) {
104         ids[i + NUM_PRODUCERS] = i;
105         pthread_create(&consumers[i], NULL, consumer_binary, &ids[i
                + NUM_PRODUCERS]);
106     }
107
108     for (int i = 0; i < NUM_PRODUCERS; i++) {
109         pthread_join(producers[i], NULL);
110     }
111
112     for (int i = 0; i < NUM_CONSUMERS; i++) {
113         pthread_join(consumers[i], NULL);
114     }
115
116     binary_semaphore_destroy(&empty);
117     binary_semaphore_destroy(&full);
118     return 0;
119 }
```

A continuación, se presenta un ejemplo de uso de un semáforo de conteo. El ejemplo muestra cómo varios hilos pueden incrementar o decrementar un contador compartido usando el semáforo de conteo para asegurar la sincronización entre ellos.

2.2.4. Funciones del ejemplo del semáforo de conteo

- **Función increment_counter:**

```
1  void* increment_counter(void* arg) {
2      int id = *((int*) arg);
```

```
3       for (int i = 0; i < INCREMENT_COUNT; i++) {
4           counting_semaphore_wait(&counting_semaphore);
5           if (shared_counter < 100) {
6               shared_counter++;
7               printf("Thread %d incremented counter to %d\n", id
                    , shared_counter);
8           }
9           counting_semaphore_post(&counting_semaphore);
10          sleep(1);
11
12          if (shared_counter == STOP_VALUE) {
13              break;
14          }
15      }
16      return NULL;
17  }
```

Esta función incrementa un contador compartido usando un semáforo de conteo. Espera el semáforo antes de incrementar el contador y lo libera después. La función se detiene si el contador alcanza un valor de parada específico.

- **Función decrement_counter:**

```
1   void* decrement_counter(void* arg) {
2       int id = *((int*) arg);
3       for (int i = 0; i < DECREMENT_COUNT; i++) {
4           counting_semaphore_wait(&counting_semaphore);
5           if (shared_counter > 0) {
6               shared_counter--;
7               printf("Thread %d decremented counter to %d\n", id
                    , shared_counter);
8           }
9           counting_semaphore_post(&counting_semaphore);
10          sleep(1);
11          if (shared_counter == STOP_VALUE) {
12              break;
13          }
14      }
15      return NULL;
16  }
```

Esta función decrementa el contador compartido usando el semáforo de conteo. Similar a la función anterior, espera el semáforo antes de decrementar el contador y lo libera después. La función se detiene si el contador alcanza el valor de parada.

- **Función main:**

```
1   #define NUM_THREADS 10
2   #define INITIAL_COUNTER 20
3   #define INCREMENT_COUNT 5
4   #define DECREMENT_COUNT 5
5   #define STOP_VALUE 20
6
7   int shared_counter = INITIAL_COUNTER;
```

```
 8  counting_semaphore_t counting_semaphore;
 9
10  int main() {
11      pthread_t threads[NUM_THREADS];
12      int ids[NUM_THREADS];
13
14      counting_semaphore_init(&counting_semaphore, 1);
15
16      for (int i = 0; i < NUM_THREADS / 2; i++) {
17          ids[i] = i;
18          pthread_create(&threads[i], NULL, increment_counter, &
                ids[i]);
19      }
20
21      for (int i = NUM_THREADS / 2; i < NUM_THREADS; i++) {
22          ids[i] = i;
23          pthread_create(&threads[i], NULL, decrement_counter, &
                ids[i]);
24      }
25
26      for (int i = 0; i < NUM_THREADS; i++) {
27          pthread_join(threads[i], NULL);
28      }
29
30      counting_semaphore_destroy(&counting_semaphore);
31
32      printf("Final counter value: %d\n", shared_counter);
33
34      return 0;
35  }
```

En esta función principal:

- Se definen y inicializan las variables necesarias, incluyendo el semáforo de conteo.
- Se crean hilos para incrementar y decrementar el contador.
- Se espera a que todos los hilos terminen su ejecución.
- Finalmente, se destruye el semáforo de conteo y se imprime el valor final del contador compartido.

2.2.5. Código Completo

A continuación, se presenta todo el código junto:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4  #include <unistd.h>
5
6  #define NUM_THREADS 10
7  #define INITIAL_COUNTER 20
8  #define INCREMENT_COUNT 5
9  #define DECREMENT_COUNT 5
10 #define STOP_VALUE 20
```

```
11
12 int shared_counter = INITIAL_COUNTER;
13
14 typedef struct {
15     pthread_mutex_t mutex;
16     pthread_cond_t condition;
17     int value;
18 } counting_semaphore_t;
19
20 void counting_semaphore_init(counting_semaphore_t *sem, int value)
       {
21     pthread_mutex_init(&sem->mutex, NULL);
22     pthread_cond_init(&sem->condition, NULL);
23     sem->value = value;
24 }
25
26 void counting_semaphore_wait(counting_semaphore_t *sem) {
27     pthread_mutex_lock(&sem->mutex);
28     while (sem->value <= 0) {
29         pthread_cond_wait(&sem->condition, &sem->mutex);
30     }
31     sem->value--;
32     pthread_mutex_unlock(&sem->mutex);
33 }
34
35 void counting_semaphore_post(counting_semaphore_t *sem) {
36     pthread_mutex_lock(&sem->mutex);
37     sem->value++;
38     pthread_cond_signal(&sem->condition);
39     pthread_mutex_unlock(&sem->mutex);
40 }
41
42 void counting_semaphore_destroy(counting_semaphore_t *sem) {
43     pthread_mutex_destroy(&sem->mutex);
44     pthread_cond_destroy(&sem->condition);
45 }
46
47 counting_semaphore_t counting_semaphore;
48
49 void* increment_counter(void* arg) {
50     int id = *((int*) arg);
51     for (int i = 0; i < INCREMENT_COUNT; i++) {
52         counting_semaphore_wait(&counting_semaphore);
53         if (shared_counter < 100) {
54             shared_counter++;
55             printf("Thread %d incremented counter to %d\n", id,
                       shared_counter);
56         }
57         counting_semaphore_post(&counting_semaphore);
58         sleep(1);
59
60         if (shared_counter == STOP_VALUE) {
61             break;
62         }
63     }
64     return NULL;
65 }
```

```
66
67  void* decrement_counter(void* arg) {
68      int id = *((int*) arg);
69      for (int i = 0; i < DECREMENT_COUNT; i++) {
70          counting_semaphore_wait(&counting_semaphore);
71          if (shared_counter > 0) {
72              shared_counter--;
73              printf("Thread %d decremented counter to %d\n", id,
                    shared_counter);
74          }
75          counting_semaphore_post(&counting_semaphore);
76          sleep(1);
77          if (shared_counter == STOP_VALUE) {
78              break;
79          }
80      }
81      return NULL;
82  }
83
84  int main() {
85      pthread_t threads[NUM_THREADS];
86      int ids[NUM_THREADS];
87
88      counting_semaphore_init(&counting_semaphore, 1);
89
90      for (int i = 0; i < NUM_THREADS / 2; i++) {
91          ids[i] = i;
92          pthread_create(&threads[i], NULL, increment_counter, &ids[i
                ]);
93      }
94
95      for (int i = NUM_THREADS / 2; i < NUM_THREADS; i++) {
96          ids[i] = i;
97          pthread_create(&threads[i], NULL, decrement_counter, &ids[i
                ]);
98      }
99
100     for (int i = 0; i < NUM_THREADS; i++) {
101         pthread_join(threads[i], NULL);
102     }
103
104     counting_semaphore_destroy(&counting_semaphore);
105
106     printf("Final counter value: %d\n", shared_counter);
107
108     return 0;
109 }
```

2.3. Barrera

2.3.1. Explicación del concepto

Una **barrera** es una estructura de sincronización empleada en programación paralela y concurrente la cual permite el coordinar múltiples hilos o procesos para que todos sean capaces de alcanzar un punto en concreto en ejecución antes

de poder continuar y esto es bastante útil en aplicaciones en las que se necesita asegurar que ciertas etapas de procesamiento sean completadas antes de avanzar a la siguiente fase (Desai, 2020).

Comúnmente, las barreras se utilizan cuando se quieren sincronizar fases, sobre todo en aplicaciones paralelas, ya que garantizan que permiten que todas las tareas de una fase en concreto se completen antes de que pasen a la siguiente fase. Además, permiten coordinar hilos en bucles paralelos ya que las barreras aseguran que todos los hilos terminen las iteraciones antes de pasar a la que sigue y esto también permite una mejor sincronización en algoritmos paralelos debido a que permiten sincronizar de manera frecuente hilos o procesos, siendo así que hay una mejor coordinación sin que se comuniquen de manera explícita (Desai, 2020).

Cada uno de los usos anteriores, dejan en claro que una de la principales ventajas de usar barreras es que permiten sincronizar de manera sencilla múltiples hilos o procesos en puntos cocnretos de un programa al mismo tiempo que facilitan la programación paralela debido a que simplifican el poder implementar algoritmos paralelos puesto que todas las tareas alcanzan un mismo punto de ejecución y esto reduce el riesgo de errores puesto que garantiza que todos os hilos o procesos se sincronicen en puntos en concreto, siendo así que el riesgo de condiciones de carrera es mucho más bajo (Desai, 2020).

Por otra parte, las desventajas que trae es que tienen un redimiento limitado en el caso de que un hilo o proceso sea significativamente más lesto que el resto, siendo así que se provocan cuellos de botella al esperar por la barrera. Esto se traduce en uso ineficiente de recursos, sobre todo si hay procesos inactivos hasta que se alcance la barrera por todos debido a la otra desventaja que es la dificultad para implementar esta estructura en ciertas condiciones (Desai, 2020).

Pese a las desventajas que pueden presentar, las barreras son herramientas verdaderamente útiles a la hora de sincronizar procesos e hilos en programación paralela pero deben ser manejados adecuadamente para que no hayan problemas de rendimiento (Desai, 2020).

2.3.2. Descripción de la implementación

Estructura de la barrera

```
1 typedef struct {
2     pthread_mutex_t mutex;
3     pthread_cond_t conditional;
4     int count;
5     int wait;
6 } barrier_t;
```

- La estructura barriet_t encapsula un mutex, una variable de condición, y dos contadores para gestionar la sincronización entre hilos.

Función barrier_init

```
1  void barrier_init(barrier_t *barrier, int count) {
2      pthread_mutex_init(&barrier->mutex, NULL);
3      pthread_cond_init(&barrier->conditional, NULL);
4      barrier->count = count; barrier->wait = 0;
5  }
```

- **Descripción:**
  - Inicializa la estructura de la barrera con el número de hilos que deben sincronizarse.
- **Entradas:**
  - `barrier`: Un puntero a la estructura de la barrera.
  - `count`: Número entero que indica la cantidad de hilos que deben alcanzar la barrera.
- **Salidas:**
  - Ninguna.
- **Restricciones:**
  - La variable `barrier` no debe ser `NULL` y debe apuntar a un área de memoria válida.
  - `count` debe ser mayor que 0.

Función barrier_wait

```
1  void barrier_wait(barrier_t *barrier) {
2      pthread_mutex_lock(&barrier->mutex);
3      barrier->wait++;
4      if (barrier->wait == barrier->count) {
5          pthread_cond_broadcast(&barrier->conditional);
6          barrier->wait = 0;
7      } else {
8          pthread_cond_wait(&barrier->conditional, &barrier->mutex);
9      }
10     pthread_mutex_unlock(&barrier->mutex);
11 }
```

- **Descripción:**
  - Espera a que todos los hilos alcancen la barrera. Si el número de hilos en la barrera es igual al número esperado, se desbloquean todos los hilos.

- **Entradas:**
  - `barrier`: Un puntero a la estructura de la barrera.
- **Salidas:**
  - Ninguna.
- **Restricciones:**
  - La variable barrier no debe ser NULL y debe apuntar a un área de memoria válida.

Función barrier_destroy

```
1  void barrier_destroy(barrier_t *barrier) {
2      pthread_mutex_destroy(&barrier->mutex);
3      pthread_cond_destroy(&barrier->conditional);
4  }
```

- **Descripción:**
  - Destruye la barrera, liberando los recursos asociados al mutex y la variable de condición.
- **Entradas:**
  - `barrier`: Un puntero a la estructura de la barrera.
- **Salidas:**
  - Ninguna.
- **Restricciones:**
  - La variable barrier no debe ser nula y debe apuntar a un área de memoria válida.

2.3.3. Descripción del ejemplo

- El ejemplo simula hilos que llegan a una barrera de sincronización, esperando a que todos los hilos lleguen antes de continuar su ejecución.
- Es importante aclarar que el ejemplo forma parte del archivo main.c del repositorio de GitHub, por lo que aquí se incluye únicamente lo referente al ejemplo de la barrera, pero en el main.c está integrado dentro del flujo principal del programa.
- Inclusión de bibliotecas necesarias:

```
1       #include <stdio.h>
2       #include <stdlib.h>
3       #include <pthread.h>
4       #include <unistd.h>
```

Estas líneas de código importan las bibliotecas estándar de C necesarias para manejar entradas y salidas (stdio.h), asignación de memoria (stdlib.h), creación y manejo de hilos (pthread.h), y funciones relacionadas con el tiempo (unistd.h).

- Definición de constantes y variables:

```
1       #define THREADS 10
2       barrier_t barrier;
3       pthread_t threads[THREADS];
4       int ids[THREADS];
```

Aquí se define una constante THREADS que establece el número de hilos a crear, en este caso, 10. También se declara una variable de tipo barrier_t llamada barrier para manejar la sincronización, un arreglo threads de tipo pthread_t para almacenar los identificadores de los hilos creados, y un arreglo ids para los identificadores de los hilos.

- Inicialización de la barrera:

```
1       barrier_init(&barrier, THREADS);
```

Se inicializa la barrera utilizando la función barrier_init, configurando la barrera para que espere a un número de hilos especificado (THREADS) antes de permitir que todos continúen su ejecución.

- Creación de hilos:

```
1       for (int i = 0; i < THREADS; i++) {
2           ids[i] = i;
3           pthread_create(&threads[i], NULL, barrier_example, &
                ids[i]);
4       }
```

En este bloque, se crea un bucle que itera THREADS veces para crear los hilos. En cada iteración:

  - Se asigna el valor del índice i al arreglo ids, que actúa como identificador único para cada hilo.
  - Se llama a pthread_create para crear un hilo que ejecutará la función barrier_example, pasándole como argumento el puntero al identificador del hilo.

- Espera a que terminen los hilos:

```
1       for (int i = 0; i < THREADS; i++) {
2           pthread_join(threads[i], NULL);
3       }
```

Este bloque utiliza otro bucle para esperar a que todos los hilos terminen su ejecución. pthread_join asegura que el hilo principal espere la terminación de cada hilo hijo antes de continuar, garantizando que todos los hilos hayan pasado por la barrera antes de que el programa continúe.

- Destrucción de la barrera:

```
1       barrier_destroy(&barrier);
```

Después de que todos los hilos han terminado su ejecución, la barrera se destruye usando la función barrier_destroy. Esto libera cualquier recurso del sistema asociado con la barrera, asegurando una limpieza adecuada.

- Finalización del programa principal:

```
1       return 0;
```

Finalmente, la función main retorna 0, indicando que el programa ha finalizado correctamente.

- A continuación, se encuentra todo el código sin seccionar:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4  #include <unistd.h>
5
6  #define THREADS 10
7
8  typedef struct {
9      pthread_mutex_t mutex;
10     pthread_cond_t cond;
11     int count;
12     int total;
13 } barrier_t;
14
15 barrier_t barrier;
16
17 void barrier_init(barrier_t *barrier, int total) {
18     barrier->total = total;
19     barrier->count = 0;
20     pthread_mutex_init(&barrier->mutex, NULL);
21     pthread_cond_init(&barrier->cond, NULL);
22 }
23
24 void barrier_wait(barrier_t *barrier) {
25     pthread_mutex_lock(&barrier->mutex);
26     barrier->count++;
27     if (barrier->count == barrier->total) {
28         barrier->count = 0;
29         pthread_cond_broadcast(&barrier->cond);
30     } else {
31         while (pthread_cond_wait(&barrier->cond, &barrier->mutex)
               != 0);
32     }
```

```
33      pthread_mutex_unlock(&barrier->mutex);
34  }
35
36  void barrier_destroy(barrier_t *barrier) {
37      pthread_mutex_destroy(&barrier->mutex);
38      pthread_cond_destroy(&barrier->cond);
39  }
40
41  void* barrier_example(void* arg) {
42      int id = *((int*) arg);
43      printf("Thread %d reached the barrier\n", id);
44      barrier_wait(&barrier);
45      printf("Thread %d passed the barrier\n", id);
46      return NULL;
47  }
48
49  int main() {
50      pthread_t threads[THREADS];
51      int ids[THREADS];
52
53      barrier_init(&barrier, THREADS);
54
55      for (int i = 0; i < THREADS; i++) {
56          ids[i] = i;
57          pthread_create(&threads[i], NULL, barrier_example, &ids[i])
                ;
58      }
59
60      for (int i = 0; i < THREADS; i++) {
61          pthread_join(threads[i], NULL);
62      }
63
64      barrier_destroy(&barrier);
65
66      return 0;
67  }
```

2.4. Read/Write Lock

2.4.1. Explicación del concepto

Un **Read/Write Lock**, que en español se conoce como bloqueo de lectura/escritura, es un mecanismo de sincronización que el cual brinda la posibilidad a múltiples hilos leer de forma simultánea un recurso compartido, pero esto lo hace restringiendo el acceso a un solo hilo en el momento que se necesita escribir en dicho recurso. Esto es particularmente útil en situaciones donde las operaciones de lectura ocurren en mayor medida en comparación con las de escritura (IBM, 2023b).

Por lo general, estas estructuras son utilizadas en contextos como lo son bases de datos, debido a que se llevan a cabo más lecturas que escrituras, por lo que los Read/Write locks permiten un mejor rendimiento al brindar accesos de

manera concurrente de lectura. A su vez, son usados en cachés y estructuras de datos compartidas en memoria, puesto que permiten acceder de manera concurrente de forma de lectura mientras que la escritura es bloqueada. Finalmente, en sistemas de archivos permiten controlar el acceso concurrente, de tal forma que se permiten varias operaciones de lectura pero se limitan las de escritura para que haya una mejor consistencia de datos (rsaxvc.net, 2012).

Al tener varias aplicaciones, estas estructuras brindan una serie de ventajas como es la mejora del rendimiento en lecturas concurrentes al permitir que varios hilos sean capaces de leer al mismo tiempo, lo cual es ideal en ocasiones donde se realizan operaciones de lectura de forma frecuente mientras que las de escritura no tanto. Además, brindan una mayor flexibilidad al dar un equilibrio entre la exclusión mutua para operaciones de escritura y la capacidad de poder tener concurrencia en operaciones de lectura. Por otro lado, esta estructura no bloquea del todo el recurso para las operaciones de lectura, por lo que se reducen los bloqueos innecesarios (rsaxvc.net, 2012).

Ahora bien, aunque presentan varias ventajas, también tienen desventajas, siendo una de las más claras su complejidad de implementación, pues en sistemas con muchos hilos o procesos resultan más difíciles de implementar a comparación de un mutex. Es también ineficiente cuando hay escrituras frecuentes, por lo que en esos casos no se debería de aplicar. A su vez, pueden surgir condiciones de inanición en el caso que los escritores deban esperar de forma indefinida en el caso de que haya un flujo constante de lectores si no se implementa bien (rsaxvc.net, 2012).

Por lo tanto, los Read/Write Locks son bastante útiles cuando se busca mejorar la concurrencia en sistemas que predominan las operaciones de lectura pero necesitan de un manejo más cuidado para que se eviten problemas de rendimiento y prioridad (rsaxvc.net, 2012).

2.4.2. Descripción de la implementación

Estructura del candado de lectura/escritura

```
1  typedef struct {
2      pthread_mutex_t mutex;
3      pthread_cond_t read;
4      pthread_cond_t write;
5      int readers;
6      int writers;
7      int writing_flag;
8  } read_write_lock_t;
```

- La estructura `read_write_lock_t` encapsula un mutex, dos variables de condición y contadores para gestionar la sincronización entre lectores y escritores.

Función `read_write_lock_init`

```
1  void read_write_lock_init(read_write_lock_t *read_write_lock) {
2      pthread_mutex_init(&read_write_lock->mutex, NULL);
3      pthread_cond_init(&read_write_lock->read, NULL);
4      pthread_cond_init(&read_write_lock->write, NULL);
5      read_write_lock->readers = 0;
6      read_write_lock->writers = 0;
7      read_write_lock->writing_flag = false;
8  }
```

- **Descripción:**
  - Inicializa la estructura del candado de lectura/escritura con un mutex, dos variables de condición, y contadores inicializados a cero.
- **Entradas:**
  - `read_write_lock`: Un puntero a la estructura del candado de lectura/escritura.
- **Salidas:**
  - Ninguna.
- **Restricciones:**
  - La variable `read_write_lock` no debe ser `NULL` y debe apuntar a un área de memoria válida.

Función `read_write_lock_read_lock`

```
1  void read_write_lock_read_lock(read_write_lock_t *read_write_lock)
       {
2      pthread_mutex_lock(&read_write_lock->mutex);
3
4      while (read_write_lock->writing_flag) {
5          pthread_cond_wait(&read_write_lock->read, &read_write_lock
               ->mutex);
6      }
7
8      read_write_lock->readers++;
9      pthread_mutex_unlock(&read_write_lock->mutex);
10 }
```

- **Descripción:**
  - Bloquea el candado de lectura, permitiendo que múltiples hilos lean simultáneamente si no hay escritores activos.
- **Entradas:**

- `read_write_lock`: Un puntero a la estructura del candado de lectura/escritura.

- **Salidas:**
  - Ninguna.

- **Restricciones:**
  - La variable `read_write_lock` no debe ser `NULL` y debe apuntar a un área de memoria válida.
  - Se asume que el candado ha sido previamente inicializado con `read_write_lock_init`.

Función `read_write_lock_write_lock`

```
1  void read_write_lock_write_lock(read_write_lock_t *read_write_lock)
       {
2      pthread_mutex_lock(&read_write_lock->mutex);
3      read_write_lock->writers++;
4
5      while (read_write_lock->readers > 0) {
6          pthread_cond_wait(&read_write_lock->write, &read_write_lock
               ->mutex);
7      }
8
9      read_write_lock->writers--;
10     read_write_lock->writing_flag = true;
11     pthread_mutex_unlock(&read_write_lock->mutex);
12 }
```

- **Descripción:**
  - Bloquea el candado de escritura, garantizando exclusividad para un único hilo escritor.

- **Entradas:**
  - `read_write_lock`: Un puntero a la estructura del candado de lectura/escritura.

- **Salidas:**
  - Ninguna.

- **Restricciones:**
  - La variable `read_write_lock` no debe ser `NULL` y debe apuntar a un área de memoria válida.
  - Se asume que el candado ha sido previamente inicializado con `read_write_lock_init`.

Función `read_write_lock_unlock`

```
1  void read_write_lock_unlock(read_write_lock_t *read_write_lock) {
2      pthread_mutex_lock(&read_write_lock->mutex);
3      if (read_write_lock->writing_flag) {
4          read_write_lock->writing_flag = false;
5          pthread_cond_broadcast(&read_write_lock->read);
6          pthread_cond_signal(&read_write_lock->write);
7      } else {
8          read_write_lock->readers--;
9          if ((read_write_lock->readers == 0) && (read_write_lock->
               writers > 0)) {
10             pthread_cond_signal(&read_write_lock->write);
11         }
12     }
13     pthread_mutex_unlock(&read_write_lock->mutex);
14 }
```

- **Descripción:**
  - Libera el candado de lectura o escritura, permitiendo que otros hilos continúen.
- **Entradas:**
  - `read_write_lock`: Un puntero a la estructura del candado de lectura/escritura.
- **Salidas:**
  - Ninguna.
- **Restricciones:**
  - La variable `read_write_lock` no debe ser `NULL` y debe apuntar a un área de memoria válida.
  - Se asume que el candado ha sido previamente inicializado con `read_write_lock_init` y bloqueado por el hilo actual.

Función `read_write_lock_destroy`

```
1  void read_write_lock_destroy(read_write_lock_t *read_write_lock) {
2      pthread_mutex_destroy(&read_write_lock->mutex);
3      pthread_cond_destroy(&read_write_lock->read);
4      pthread_cond_destroy(&read_write_lock->write);
5  }
```

- **Descripción:**
  - Destruye el candado de lectura/escritura, liberando los recursos asociados.

- **Entradas:**
  - `read_write_lock`: Un puntero a la estructura del candado de lectura/escritura.
- **Salidas:**
  - Ninguna.
- **Restricciones:**
  - La variable `read_write_lock` no debe ser `NULL` y debe apuntar a un área de memoria válida.
  - Se asume que el candado ha sido previamente inicializado con `read_write_lock_init`.

2.4.3. Descripción del Ejemplo

- El ejemplo simula un sistema donde múltiples hilos lectores leen de un buffer compartido, utilizando un candado de lectura/escritura para gestionar el acceso concurrente.
- Inclusión de bibliotecas necesarias:

```
1     #include <stdio.h>
2     #include <stdlib.h>
3     #include <pthread.h>
4     #include <unistd.h>
5     #include <stdbool.h>
6     #include <string.h>
```

  Estas líneas de código importan las bibliotecas estándar de C necesarias para manejar entradas y salidas (stdio.h), asignación de memoria (stdlib.h), creación y manejo de hilos (pthread.h), y funciones relacionadas con el tiempo (unistd.h). Además, se usa stdbool para booleanos y string para poder manejar los strings en el ejemplo.

- Definición de constantes y variables:

```
1     #define BUFFER_LENGTH 1024
2     #define NUM_STRINGS 4
3     #define NUM_READERS 5
4
5     int current_message_index = 0;
6     int finished_writing = 0;
7
8     char shared_buffer[BUFFER_LENGTH] = {0};
9     char *messages[NUM_STRINGS] = {
10        "Cadena de texto #1",
11        "Segunda cadena de texto",
12        "Ahora es la tercera cadena de texto",
13        "Cadena de texto final del ejemplo",
14    };
15
16    read_write_lock_t read_write_lock;
```

Aquí se definen la longitud del buffer compartido y el número de cadenas de texto. También se declaran el buffer compartido y las cadenas de texto. Además, se declara una variable de tipo `read_write_lock_t` llamada `read_write_lock` para manejar la sincronización.

- Implementación de la función para bloquear el candado de lectura:

```
1       void read_write_lock_init(read_write_lock_t *
            read_write_lock) {
2           pthread_mutex_init(&read_write_lock->mutex, NULL);
3           pthread_cond_init(&read_write_lock->read, NULL);
4           pthread_cond_init(&read_write_lock->write, NULL);
5           read_write_lock->readers = 0;
6           read_write_lock->writers = 0;
7           read_write_lock->writing_flag = false;
8       }
```

Esta función inicializa el candado de lectura/escritura, configurando el mutex y las variables de condición, y estableciendo los contadores de lectores y escritores.

- Implementación de las funciones para bloquear y desbloquear el candado de lectura y escritura:

```
1       void read_write_lock_read_lock(read_write_lock_t *
            read_write_lock) {
2           pthread_mutex_lock(&read_write_lock->mutex);
3
4           while (read_write_lock->writing_flag) {
5               pthread_cond_wait(&read_write_lock->read, &
                    read_write_lock->mutex);
6           }
7
8           read_write_lock->readers++;
9           pthread_mutex_unlock(&read_write_lock->mutex);
10      }
11
12      void read_write_lock_write_lock(read_write_lock_t *
            read_write_lock) {
13          pthread_mutex_lock(&read_write_lock->mutex);
14          read_write_lock->writers++;
15
16          while (read_write_lock->readers > 0) {
17              pthread_cond_wait(&read_write_lock->write, &
                    read_write_lock->mutex);
18          }
19
20          read_write_lock->writers--;
21          read_write_lock->writing_flag = true;
22          pthread_mutex_unlock(&read_write_lock->mutex);
23      }
24
25      void read_write_lock_unlock(read_write_lock_t *
            read_write_lock) {
26          pthread_mutex_lock(&read_write_lock->mutex);
27          if (read_write_lock->writing_flag) {
```

```
28              read_write_lock->writing_flag = false;
29              pthread_cond_broadcast(&read_write_lock->read);
30              pthread_cond_signal(&read_write_lock->write);
31          } else {
32              read_write_lock->readers--;
33              if ((read_write_lock->readers == 0) && (
                    read_write_lock->writers > 0)) {
34                  pthread_cond_signal(&read_write_lock->write);
35              }
36          }
37          pthread_mutex_unlock(&read_write_lock->mutex);
38      }
39
40      void read_write_lock_destroy(read_write_lock_t *
            read_write_lock) {
41          pthread_mutex_destroy(&read_write_lock->mutex);
42          pthread_cond_destroy(&read_write_lock->read);
43          pthread_cond_destroy(&read_write_lock->write);
44      }
```

Estas funciones implementan el bloqueo y desbloqueo del candado de lectura y escritura, así como la destrucción del candado.

- Función de hilo lector:

```
1       void* reader_thread_function(void* arg) {
2           long thread_id = (long)arg;
3           char last_message[BUFFER_LENGTH] = "";
4
5           while (1) {
6               read_write_lock_read_lock(&read_write_lock);
7
8               if (finished_writing && current_message_index >=
                    NUM_STRINGS) {
9                   read_write_lock_unlock(&read_write_lock);
10                  break;
11              }
12
13              if (strcmp(shared_buffer, last_message) != 0) {
14                  printf("Reader %ld: %s\n", thread_id,
                        shared_buffer);
15                  strcpy(last_message, shared_buffer);
16              }
17
18              read_write_lock_unlock(&read_write_lock);
19              usleep(250000);
20          }
21          return NULL;
22      }
```

Esta función es ejecutada por los hilos lectores, que leen el buffer compartido mientras el candado de lectura está bloqueado para evitar conflictos con escritores. Utiliza la función `read_write_lock_read_lock` para obtener acceso de lectura y `read_write_lock_unlock` para liberar el candado.

- Función para copiar lentamente cadenas:

```
1       void slow_copy(char *destination, char *origin, int length
            ) {
2           for (int i = 0; i < length; i++) {
3               destination[i] = origin[i];
4               if (destination[i] == '\0') break;
5               usleep(50000);
6           }
7       }
```

Esta función copia lentamente una cadena de caracteres de `origin` a `destination`, con una pausa entre cada carácter.

- Código completo:

```
1       #include <stdio.h>
2       #include <stdlib.h>
3       #include <pthread.h>
4       #include <unistd.h>
5       #include <stdbool.h>
6       #include <string.h>
7
8       #define BUFFER_LENGTH 1024
9       #define NUM_STRINGS 4
10      #define NUM_READERS 5
11
12      typedef struct {
13          pthread_mutex_t mutex;
14          pthread_cond_t read;
15          pthread_cond_t write;
16          int readers;
17          int writers;
18          bool writing_flag;
19      } read_write_lock_t;
20
21      char shared_buffer[BUFFER_LENGTH] = {0};
22      char *messages[NUM_STRINGS] = {
23          "Cadena de texto #1",
24          "Segunda cadena de texto",
25          "Ahora es la tercera cadena de texto",
26          "Cadena de texto final del ejemplo",
27      };
28
29      read_write_lock_t read_write_lock;
30      bool finished_writing = false;
31      int current_message_index = 0;
32
33      void read_write_lock_init(read_write_lock_t *
            read_write_lock) {
34          pthread_mutex_init(&read_write_lock->mutex, NULL);
35          pthread_cond_init(&read_write_lock->read, NULL);
36          pthread_cond_init(&read_write_lock->write, NULL);
37          read_write_lock->readers = 0;
38          read_write_lock->writers = 0;
39          read_write_lock->writing_flag = false;
40      }
41
42      void read_write_lock_read_lock(read_write_lock_t *
            read_write_lock) {
```

```
43          pthread_mutex_lock(&read_write_lock->mutex);
44          while (read_write_lock->writing_flag) {
45              pthread_cond_wait(&read_write_lock->read, &
                    read_write_lock->mutex);
46          }
47          read_write_lock->readers++;
48          pthread_mutex_unlock(&read_write_lock->mutex);
49      }
50
51      void read_write_lock_write_lock(read_write_lock_t *
            read_write_lock) {
52          pthread_mutex_lock(&read_write_lock->mutex);
53          read_write_lock->writers++;
54          while (read_write_lock->readers > 0) {
55              pthread_cond_wait(&read_write_lock->write, &
                    read_write_lock->mutex);
56          }
57          read_write_lock->writers--;
58          read_write_lock->writing_flag = true;
59          pthread_mutex_unlock(&read_write_lock->mutex);
60      }
61
62      void read_write_lock_unlock(read_write_lock_t *
            read_write_lock) {
63          pthread_mutex_lock(&read_write_lock->mutex);
64          if (read_write_lock->writing_flag) {
65              read_write_lock->writing_flag = false;
66              pthread_cond_broadcast(&read_write_lock->read);
67              pthread_cond_signal(&read_write_lock->write);
68          } else {
69              read_write_lock->readers--;
70              if (read_write_lock->readers == 0 &&
                    read_write_lock->writers > 0) {
71                  pthread_cond_signal(&read_write_lock->write);
72              }
73          }
74          pthread_mutex_unlock(&read_write_lock->mutex);
75      }
76
77      void read_write_lock_destroy(read_write_lock_t *
            read_write_lock) {
78          pthread_mutex_destroy(&read_write_lock->mutex);
79          pthread_cond_destroy(&read_write_lock->read);
80          pthread_cond_destroy(&read_write_lock->write);
81      }
82
83      void* reader_thread_function(void* arg) {
84          long thread_id = (long)arg;
85          char last_message[BUFFER_LENGTH] = "";
86
87          while (1) {
88              read_write_lock_read_lock(&read_write_lock);
89
90              if (finished_writing && current_message_index >=
                    NUM_STRINGS) {
91                  read_write_lock_unlock(&read_write_lock);
92                  break;
```

```
93              }
94
95              if (strcmp(shared_buffer, last_message) != 0) {
96                  printf("Reader %ld: %s\n", thread_id,
                        shared_buffer);
97                  strcpy(last_message, shared_buffer);
98              }
99
100             read_write_lock_unlock(&read_write_lock);
101             usleep(250000);
102         }
103         return NULL;
104     }
105
106     void slow_copy(char *destination, char *origin, int length
        ) {
107         for (int i = 0; i < length; i++) {
108             destination[i] = origin[i];
109             if (destination[i] == '\0') break;
110             usleep(50000);
111         }
112     }
113
114     int main() {
115         pthread_t readers[NUM_READERS];
116
117         read_write_lock_init(&read_write_lock);
118
119         for (long i = 0; i < NUM_READERS; i++) {
120             pthread_create(&readers[i], NULL,
                    reader_thread_function, (void *)i);
121         }
122
123         while (current_message_index < NUM_STRINGS) {
124             read_write_lock_write_lock(&read_write_lock);
125             slow_copy(shared_buffer, messages[
                    current_message_index], BUFFER_LENGTH);
126             read_write_lock_unlock(&read_write_lock);
127             current_message_index++;
128             sleep(2);
129         }
130
131         read_write_lock_write_lock(&read_write_lock);
132         finished_writing = 1;
133         pthread_cond_broadcast(&read_write_lock.read);
134         read_write_lock_unlock(&read_write_lock);
135
136         for (int i = 0; i < NUM_READERS; i++) {
137             pthread_join(readers[i], NULL);
138         }
139
140         read_write_lock_destroy(&read_write_lock);
141
142         return 0;
143     }
```

En el código completo se incluyen todas las funciones necesarias para la

implementación del candado de lectura/escritura y el ejemplo de uso de hilos lectores y escritores.

Capítulo 3

Conclusiones

- La correcta implementación de mutexes, semáforos, barreras y read/write locks en C utilizando pthreads permite que haya una mejor sincronización efectiva entre hilos, evitando así que sucedan problemas como es el caso de condiciones de carrera o acceso no controlado a recursos compartidos pero esto es únicamente cuando se hacen implementaciones adecuadas a los contextos en los que se solicitan.

- Un mutex brinda exclusividad para acceder a secciones críticas, permitiendo así que solo un hilo sea capaz de acceder a una región de código o recurso a la vez, lo que es vital para que se pueda tener una integridad en los datos compartidos.

- Los semáforos brindan la posibilidad de tener un control sobre el acceso a recursos limitados de tal forma que facilitan la gestión en los casos en los que múltiples hilos tengan que esperar o proceder en base a qué tan disponibles se encuentren los recursos, de tal forma que se evita que exista una sobrecarga y bloqueos que no son necesarios.

- Las barreras son sumamente efectivas cuando se trata sincronizar el progreso de varios hilos, de tal forma que se garantiza que todos los hilos lleguen a un punto en concreto antes de poder continuar, lo cual es de gran utilidad cuando se trabajan algoritmos que necesitan de fases sincronizadas entre múltiples hilos.

- Los read/write locks permiten mejorar el acceso de forma concurrente a recursos compartidos en los cuales se tienen que permitir varios lectores a la vez mientras que se encargan de controlar el acceso de forma exclusiva para escritores, de tal forma que brindan un mejor rendimiento cuando

las operaciones de lectura son mucho más frecuentes que las de escritura.

- Se debe elegir la estructura de sincronización correcta de acuerdo a la necesidad de sincronización que surja, debido que así se evitan problemas de rendimiento y sincronización en aplicaciones multihilo. Es por esto que es necesario que se conozcan las ventajas y desventajas de cada tipo, para que se puedan hacer implementaciones correctas que permitan la creación de software mucho más eficiente y estable.

Capítulo 4

Enlace al repositorio de GitHub

En el siguiente enlace se encuentra el repositorio de GitHub con todo el código de la implementación así como el manual de usuario en el README para su adecuada ejecución: `https://github.com/ch0so/biblioteca_sync`

Referencias

Brooks, N. (2024, 12 de Agosto). *What is semaphore? counting, binary types with example.* Guru99. Descargado de https://www.guru99.com/semaphore-in-operating-system.html

Casero, A. (2024, 15 de Marzo). *¿cómo funciona la sincronización en programación?* keepcoding. Descargado de https://keepcoding.io/blog/sincronizacion-en-programacion/

Desai, J. (2020, 11 de Junio). *Barrier synchronization in threads.* Medium. Descargado de https://medium.com/@jaydesai36/barrier-synchronization-in-threads-3c56f947047

Fernando. (2011, 28 de Diciembre). *Conccurrencia-mutex.* blogspot. Descargado de https://cortesfernando.blogspot.com/2011/12/concurrencia-mutex.html

GeeksforGeeks. (2023, 9 de Mayo). *Thread functions in c/c++.* GeeksforGeeks. Descargado de https://www.geeksforgeeks.org/thread-functions-in-c-c/

GeeksforGeeks. (2024, 1 de Agosto). *Semaphores in process synchronization.* GeeksforGeeks. Descargado de https://www.geeksforgeeks.org/semaphores-in-process-synchronization/

IBM. (2023a, 24 de Marzo). *Using mutexes.* IBM. Descargado de https://www.ibm.com/docs/pt-br/aix/7.2?topic=programming-using-mutexes

IBM. (2023b, 24 de Marzo). *Using read/write locks.* IBM. Descargado de https://www.ibm.com/docs/en/aix/7.2?topic=programming-using-readwrite-locks

rsaxvc.net. (2012, 29 de Septiembre). *Advantages of reader/writer locks on small single-core systems.* rsaxvc.net. Descargado de https://rsaxvc.net/blog/2012/9/23/Advantages_of_ReaderWriter_locks_on_small_single-core_systems.html