Universidad Nacional de San Agustín de Arequipa

Escuela Profesional de Ciencia de la Computación

Computación paralela y distribuida

Laboratorio: Pthreads

Presentado por

Villarroel Ramos Fiorela Estefany

Docente

Alvaro Henry Mamani Aliaga

Barriers Implementación: https://github.com/VILLA7523/PARALELA

1. Busy-waiting and a mutex

El término *busy-waiting* se refiere a una técnica en programación donde un hilo (o proceso) espera activamente en un bucle hasta que se cumple cierta condición. En el contexto de un mutex, esto significa que un hilo espera repetidamente verificando constantemente si un recurso compartido está disponible para su acceso. Cuando se utiliza busy-waiting con un mutex, un hilo espera en un bucle hasta que el mutex asociado al recurso compartido se desbloquea. Si el mutex está bloqueado por otro hilo, el hilo que espera continúa en un bucle, revisando periódicamente si el mutex se ha liberado.

Esta técnica tiene algunos inconvenientes, ya que puede llevar a un uso ineficiente de los recursos del sistema, ya que el hilo está activamente revisando constantemente el estado del mutex en lugar de ser notificado cuando el recurso está disponible. Además, en sistemas con múltiples núcleos, el busy-waiting puede competir por el tiempo de CPU con otros procesos y, por lo tanto, no es la solución más eficiente.

1.1. Implementación

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

int counter = 0;            // Inicializar a 0
int thread_count;
pthread_mutex_t barrier_mutex;

void* ThreadWork() {
    // Trabajo del hilo...
    int timer = 0;
    // Barrera
    pthread_t tid = pthread_self();
    printf("Empece: %lu\n", tid);

    pthread_mutex_lock(&barrier_mutex);
    printf("Sume counter:%lu\n", tid);
    counter++;
    pthread_mutex_unlock(&barrier_mutex);

    printf("Pase counter:%lu\n", tid);
    while (counter < thread_count) {
```

```c
        printf("Wait:%lu ", tid);
    }
    return NULL;
}

int main() {
    // Configuración inicial...

    pthread_t* thread_handles;
    thread_count = 5;  // Número de hilos

    thread_handles = (pthread_t*)malloc(thread_count * sizeof(pthread_t));

    // Crear hilos
    for (int thread = 0; thread < thread_count; thread++) {
        pthread_create(&thread_handles[thread], NULL, ThreadWork, NULL);
    }
    // Unirse a los hilos
    for (int thread = 0; thread < thread_count; thread++) {
        pthread_join(thread_handles[thread], NULL);
    }
    free(thread_handles);
    return 0;
}
```

1.2. Ejecución

```
D:\UNSA\Barriers\barriers>a

Empece: 1

Sume counter:1

Pase counter:1
Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1
Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1
Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1
Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1
Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1
Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1
Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1
Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1
Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1
Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1
Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1
Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1
Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1
Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1
Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1
Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1
Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1
```

```
Empece: 3

Sume counter:3

Pase counter:3
Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3
Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3
Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3
Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3
Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3
Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3
Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3
Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3
Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3
Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1
Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1
Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1
Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1
Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1
Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1
Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1
Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:1 Wait:3 Wait:3
Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3
Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3 Wait:3
```

```
Empece: 2

Sume counter:2

Pase counter:2
Wait:3 Wait:1
D:\UNSA\Barriers\barriers>gcc mutex.c -pthread
```

1.3. Análisis

1. Inicialización: Se inicializa counter y se especifica la cantidad de hilos (thread_count) que deben llegar a la barrera.
2. Barrera con Mutex: Cada hilo incrementa counter de manera segura utilizando un pthread_mutex para evitar condiciones de carrera.
3. Luego, entra en un bucle mientras espera a que todos los hilos lleguen a la barrera. Dentro de este bucle, se imprime "Wait"junto con el ID del hilo.
4. Ejecución de Trabajo del Hilo: La función ThreadWork representa el trabajo que realiza cada hilo antes de llegar a la barrera.
5. Impresiones: Se han agregado algunas impresiones para mostrar el flujo de ejecución y cómo los hilos interactúan con la barrera.

2. Semaphores

Los semáforos son una herramienta de sincronización que se utiliza para coordinar el acceso a recursos compartidos en entornos de programación multihilo. En pthreads (POSIX Threads), puedes utilizar la biblioteca de semáforos proporcionada por POSIX para implementar semáforos en C.

2.1. Implementación

2.2. Ejecución

```
D:\UNSA\Barriers\barriers>gcc semaphores.c -pthread

D:\UNSA\Barriers\barriers>a
Hilo 1 esta esperando
Hilo 2 esta esperando
Hilo 3 esta esperando
Hilo 4 libera a los demas
Hilo 4 ha terminado
Hilo 1 ha terminado
Hilo 2 ha terminado
Hilo 3 ha terminado
```

2.3. Análisis

- En la implementación de la barrera con semáforos utilizamos un contador para determinar cuántos hilos han ingresado a la barrera. Utilizamos dos semáforos: count_sem protege el contador, y barrier_sem se utiliza para bloquear los hilos que han ingresado a la barrera.
- El semáforo count_sem se inicializa en 1 (es decir, "desbloqueado"), por lo que el primer hilo en alcanzar la barrera podrá continuar más allá de la llamada a sem_wait. Sin embargo, los hilos siguientes se bloquearán hasta que puedan tener acceso exclusivo al contador. Cuando un hilo tiene acceso exclusivo al contador, verifica si counter <thread_count-1. Si es así, el hilo incrementa el contador, çede el bloqueo"(sem_post(count_sem)) y se bloquea en sem_wait(barrier_sem).
- Por otro lado, si counter == thread_count-1, el hilo es el último en ingresar a la barrera, por lo que puede restablecer counter a cero y "desbloquearçount_sem llamando a sem_post(count_sem). Ahora, desea notificar a todos los demás hilos que pueden continuar, así que ejecuta sem_post(barrier_sem) para cada uno de los thread_count-1 hilos bloqueados en sem_wait(barrier_sem).
- Es importante destacar que no importa si el hilo que ejecuta el bucle de llamadas a sem_post(barrier_sem) avanza y realiza múltiples llamadas a sem_post antes de que un hilo pueda desbloquearse de sem_wait(barrier_sem). Esto se debe a que eventualmente los hilos bloqueados verán que barrier_sem es positivo, lo decrementarán y procederán.
- Los hilos no necesitan consumir ciclos de CPU cuando están bloqueados en sem_wait. Además, se puede reutilizar el contador y count_sem para ejecutar una segunda barrera.
- El problema radica en reutilizar barrier_sem debido a una posible condición de carrera.

3. Condition variables

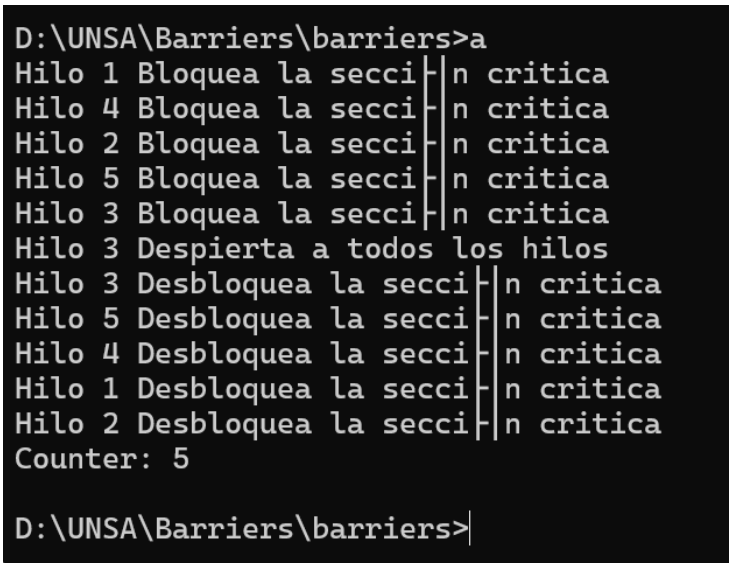
Una çondition variable"(variable de condición) en Pthreads es un objeto de datos que permite a un hilo suspender su ejecución hasta que ocurra cierto evento o condición. Cuando el evento o condición tiene lugar, otro hilo puede señalar al hilo en espera para "despertarlo". Cabe destacar que una variable de condición siempre está asociada con un "mutex".

3.1. Implementación

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

int counter = 0;
int thread_count;
pthread_mutex_t mutex;
pthread_cond_t cond_var;

void* ThreadWork(void* value) {
    pthread_t tid = pthread_self();

    // Trabajo del hilo...
    pthread_mutex_lock(&mutex);
    printf("Hilo %d Bloquea la sección critica\n" , tid);
    counter++;
    if (counter == thread_count) {
        printf("Hilo %d Despierta a todos los hilos\n" , tid);
        pthread_cond_broadcast(&cond_var);
    } else {
        while (pthread_cond_wait(&cond_var, &mutex) != 0) {
            printf("Hilo %d Espera aqui , hasta que todos lleguen\n" ,
            ↪ tid);
        }
    }
    pthread_mutex_unlock(&mutex);
    printf("Hilo %d Desbloquea la sección critica\n" , tid);
}

int main() {
    // Configuración inicial...
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&cond_var, NULL);
```

```
33
34      pthread_t* thread_handles;
35      thread_count = 5;  // Número de hilos
36
37      thread_handles = (pthread_t*)malloc(thread_count *
        ↪  sizeof(pthread_t));
38
39      // Crear hilos
40      for (int thread = 0; thread < thread_count; thread++) {
41          pthread_create(&thread_handles[thread], NULL, ThreadWork,
            ↪  (void*)&thread);
42      }
43
44      // Unirse a los hilos
45      for (int thread = 0; thread < thread_count; thread++) {
46          pthread_join(thread_handles[thread], NULL);
47      }
48
49      free(thread_handles);
50
51      // Limpieza y finalización...
52      pthread_mutex_destroy(&mutex);
53      pthread_cond_destroy(&cond_var);
54
55      printf("Counter: %d\n", counter);
56
57      return 0;
58  }
```

3.2. Ejecución



3.3. Análisis

- Cuando un hilo alcanza la barrera, incrementa un contador protegido por un mutex. Si es el último hilo en llegar, despierta a todos los hilos bloqueados en la variable de condición mediante pthread_cond_broadcast. Si no es el último hilo, el hilo se bloquea en pthread_cond_wait, liberando el mutex y esperando a ser despertado.
- El texto destaca que se debe tener precaución con posibles eventos que puedan desbloquear hilos aparte de la llamada a pthread_cond_broadcast. Por eso, la llamada a pthread_cond_wait se coloca en un bucle while, y si el hilo se despierta por un evento diferente, volverá a bloquearse.
- Además, se sugiere verificar la condición antes de proceder, especialmente si solo un hilo es despertado. En el caso de pthread_cond_broadcast, se menciona la posibilidad de carreras, ya que un hilo podría cambiar la condición después de despertar, lo que podría hacer que otros hilos se vuelvan a bloquear.
- Es crucial que pthread_cond_wait desbloquee el mutex para evitar que solo un hilo entre a la barrera y cause el bloqueo de los demás. También se destaca que el mutex debe volverse a bloquear después de la llamada a pthread_cond_wait.
- Finalmente, se menciona la necesidad de inicializar y destruir correctamente las variables de condición mediante pthread_cond_init y pthread_cond_destroy.
- Los hilos no consumen ciclos de CPU mientras esperan en pthread_cond_wait.

1. `pthread_t tid = pthread_self();`: Obtiene el identificador único del hilo actual.
2. `pthread_mutex_lock(&mutex);`: Bloquea el mutex para garantizar la exclusión mutua.

3. `printf("Hilo%d Bloquea la sección crítica\n", tid);`: Imprime un mensaje de bloqueo.
4. `counter++;`: Incrementa el contador.
5. `if (counter == thread_count) { pthread_cond_broadcast(&cond_var); }`: Despierta a todos los hilos si todos han llegado.
6. `else { while (pthread_cond_wait(&cond_var, &mutex) != 0); }`: Hilo espera en la variable de condición si no todos han llegado.
7. `pthread_mutex_unlock(&mutex);`: Libera el mutex.
8. `printf("Hilo%d Desbloquea la sección crítica\n", tid);`: Imprime un mensaje de desbloqueo.

Lista enlazada multithreding

Funciones:

- **La función Member:** Se encarga de buscar un valor deseado en una lista enlazada utilizando un puntero para recorrerla. En este caso, la lista está ordenada. El puntero se desplaza a través de los nodos de la lista hasta encontrar el valor deseado o determinar que dicho valor no puede estar presente en la lista. La condición de salida del bucle se activa cuando el puntero actual, denominado p, alcanza el final de la lista, es decir, cuando se vuelve NULL. También se verifica otra condición: si el miembro de datos del nodo actual es mayor que el valor deseado. En ambos casos, se concluye que el valor buscado no está presente en la lista.
- **La función Insert:** Se encarga de insertar un nuevo nodo en una lista enlazada de manera ordenada. Recorre la lista hasta encontrar la posición adecuada para la inserción, y luego inserta el nuevo nodo en esa posición. Si el valor ya está presente en la lista, la función devuelve 0; de lo contrario, devuelve 1 para indicar que la inserción fue exitosa. En resumen, la función maneja la inserción ordenada de nodos en una lista enlazada.
- **La función Delete:** Se encarga de eliminar un nodo específico de una lista enlazada. Al igual que la función Insert, necesita realizar un seguimiento del predecesor del nodo actual mientras busca el nodo que se va a eliminar. Después de encontrar el nodo a eliminar, actualiza el miembro next del nodo predecesor para omitir el nodo encontrado, y luego libera la memoria del nodo eliminado. La función verifica si la eliminación fue exitosa y devuelve 1 en ese caso, o 0 si el valor a eliminar no está presente en la lista. En resumen, la función Delete maneja la eliminación de un nodo específico de una lista enlazada.

¿Qué problemas pueden ocurrir al querer ingresar a la sección crítica?

- Cuando múltiples hilos ejecutan simultáneamente las tres funciones (Member, Insert, y Delete) en una estructura de datos compartida, como una lista enlazada, pueden surgir problemas debido a las operaciones de escritura (Insert y Delete) que modifican la memoria compartida. Dado que varios hilos pueden leer simultáneamente una ubicación de memoria sin conflicto, no hay problema en que múltiples hilos ejecuten simultáneamente la función Member.
- Sin embargo, al ejecutar operaciones de escritura como Insert y Delete, pueden ocurrir problemas si intentamos ejecutar cualquiera de estas operaciones al mismo tiempo que otra operación. Por ejemplo, si un hilo está ejecutando Member(5) al mismo tiempo que otro hilo está ejecutando Delete(5), pueden surgir problemas. Si el hilo 0 informa que el elemento 5 está en la lista mientras el hilo 1 lo elimina, la información puede ser incherente.
- Para abordar este problema, una solución evidente es bloquear la lista cada vez que un hilo intenta acceder a ella. Esto asegura que solo un hilo a la vez puede realizar operaciones de escritura o lectura en la lista, evitando posibles conflictos y garantizando la consistencia de los datos compartidos. Esta estrategia de bloqueo se implementa comúnmente utilizando cerraduras (locks) o semáforos para controlar el acceso concurrente a la estructura de datos compartida.

4. One Mutex for Entire List

- Una solución es abordar el problema de acceso concurrente a la estructura de datos compartida es bloqueando la lista cada vez que un hilo intenta acceder a ella. Esto se logra utilizando un mecanismo de exclusión mutua (mutex).
- Esta solución garantiza que solo un hilo a la vez pueda realizar operaciones en la lista, evitando posibles conflictos de acceso concurrente.
- El inconveniente sería que al serializar el acceso a la lista, se pierda la oportunidad de aprovechar el paralelismo, en escenarios donde la mayoría de las operaciones son llamadas a Insert y Delete, esta solución puede ser aceptable, ya que se requeriría la serialización del acceso a la lista para la mayoría de las operaciones. Sin embargo, en casos donde la mayoría de las operaciones son llamadas a Member, esta solución puede no ser la más eficiente.

4.1. Implementación

```
1
2 int member(int value) {
3     struct list_node * curr_p = head_p;
4     while(curr_p != NULL && curr_p->data < value) curr_p = curr_p->next;
5     if(curr_p == NULL || curr_p->data > value) return 0;
```

```
6       else return 1;
7   }
8
9   int insert (int value)
10  {
11      struct list_node * curr_p = *(&head_p);
12      struct list_node * prev_p = NULL;
13      struct list_node * temp_p;
14
15      while(curr_p != NULL && curr_p->data < value) {
16          prev_p = curr_p;
17          curr_p = curr_p->next;
18      }
19
20      if(curr_p == NULL || curr_p->data > value) {
21          temp_p = malloc(sizeof(struct list_node));
22          temp_p->data = value;
23          temp_p->next = curr_p;
24          if(prev_p == NULL) *(&head_p) = temp_p;
25          else prev_p->next = temp_p;
26          return 1;
27      }else {
28          return 0; //valor igual a uno de la lista
29      }
30  }
31
32  int delete(int value) {
33      struct list_node * curr_p = *(&head_p);
34      struct list_node * prev_p = NULL;
35
36      while(curr_p != NULL && curr_p->data < value) {
37          prev_p = curr_p;
38          curr_p = curr_p->next;
39      }
40
41      if(curr_p != NULL && curr_p->data == value) {
42          if(prev_p == NULL) {
43              *(&head_p) = curr_p -> next;
44              free(curr_p);
45          }else{
46              prev_p->next = curr_p->next;
47              free(curr_p);
48          }
49          return 1;
50      }else {
51          return 0;
52      }
```

```
53 }
54
55
56 int Insert(int value) {
57     pthread_t tid = pthread_self();
58     pthread_mutex_lock(&list_mutex);
59     printf("Hilo %d bloquea la lista en funcion insert\n" , tid);
60     int a = insert(value);
61     pthread_mutex_unlock(&list_mutex);
62     printf("Hilo %d desloquea la lista en funcion insert\n" , tid);
63     return a;
64 }
65
66 int Delete(int value) {
67     pthread_t tid = pthread_self();
68     pthread_mutex_lock(&list_mutex);
69     printf("Hilo %d bloquea la lista en funcion Delete\n" , tid);
70     int a = delete(value);
71     pthread_mutex_unlock(&list_mutex);
72     printf("Hilo %d desloquea la lista en funcion Delete\n" , tid);
73     return a;
74 }
75
76 int Member(int value) {
77     pthread_t tid = pthread_self();
78     pthread_mutex_lock(&list_mutex);
79     printf("Hilo %d bloquea la lista en funcion Member\n" , tid);
80     int a = member(value);
81     pthread_mutex_unlock(&list_mutex);
82     printf("Hilo %d desbloquea la lista en funcion Member\n" , tid);
83     return a;
84 }
```

4.2. Ejecución

```
D:\UNSA\Barriers\list>gcc one_mutex_for_entire_list.c -pthread

D:\UNSA\Barriers\list>a
Hilo 1 bloquea la lista en funcion insert
Hilo 1 desloquea la lista en funcion insert
Hilo 2 bloquea la lista en funcion insert
Hilo 2 desloquea la lista en funcion insert
Hilo 3 bloquea la lista en funcion insert
Hilo 3 desloquea la lista en funcion insert
Hilo 4 bloquea la lista en funcion insert
Hilo 4 desloquea la lista en funcion insert
Hilo 5 bloquea la lista en funcion Delete
Hilo 5 desloquea la lista en funcion Delete
Hilo 6 bloquea la lista en funcion insert
Hilo 6 desloquea la lista en funcion insert
Hilo 7 bloquea la lista en funcion Member
Hilo 7 desbloquea la lista en funcion Member
1 2 5 6
```

4.3. Análisis

- Cómo se puede ver en la consola , cada vez que un hilo empieza a ejecutar una función bloquea a todos los demas hilos para que no hagan ninguna operación en la lista , hasta que termina de ejecutar la funcion , es por eso que se muestra de forma consecutiva , que cada hilo bloquea y desbloquea en el mismo tiempo , para que otro pueda ejecutar otra funcion.
- En cada funcion orcurre lo siguiente: Bloquea el mutex antes de realizar cualquier función de la lista. imprime un mensaje indicando que el hilo ha bloqueado la lista, después Llama a la función respectiva para realizar la opración. Desbloquea el mutex después de completar la operación. Imprime un mensaje indicando que el hilo ha desbloqueado la lista y finalmente retorna el resultado de la operación delete.

5. One Mutex per Node

Para realizar un mutex por nodo , se realiza los bloqueos y desbloqueos para cada nodo que se esta consultando o al cual se esta modificando , la implementación de Member con un mutex por nodo donde cada nodo tiene su propio mutex. Cuando un hilo accede a un nodo, bloquea su mutex específico. Esto permite que múltiples hilos accedan a diferentes partes de la lista simultáneamente.

5.1. Implementación

5.2. Ejecución

```
D:\UNSA\Barriers\list>a
Member: Hilo 9 bloquea el nodo head (5) de la lista
Member: Hilo 9 desbloquea el nodo head (5) de la lista
Member: Hilo 9 desbloquea el nodo (5) de la lista
Member: Hilo 6 bloquea el nodo head (5) de la lista
Member: Hilo 6 desbloquea el nodo head (5) de la lista
Member: Hilo 6 desbloquea el nodo (5) de la lista
Member: Hilo 7 bloquea el nodo head (5) de la lista
Member: Hilo 7 bloquea el nodo (6) de la lista
Member: Hilo 7 desbloquea el nodo head (5) de la lista
Member: Hilo 7 desbloquea el nodo (5) de la lista
Member: Hilo 7 desbloquea el nodo (6) de la lista
Member: Hilo 8 bloquea el nodo head (5) de la lista
Member: Hilo 8 bloquea el nodo (6) de la lista
Member: Hilo 8 desbloquea el nodo head (5) de la lista
Member: Hilo 8 desbloquea el nodo (5) de la lista
Member: Hilo 8 bloquea el nodo (8) de la lista
Member: Hilo 8 desbloquea el nodo (6) de la lista
```

5.3. Análisis

- Bloqueo del nodo head: Se bloquea el mutex head_p_mutex para asegurar un acceso exclusivo al nodo head_p de la lista. Se imprime un mensaje indicando que el hilo ha bloqueado el nodo head.
- Inicialización del puntero temporal: Se asigna el puntero temporal temp_p para recorrer la lista, comenzando desde el nodo head_p.
- Recorrido de la lista: Se entra en un bucle que recorre la lista mientras el nodo actual (temp_p) no sea nulo y su valor sea menor que el valor deseado.
- Para cada nodo en el bucle:Se bloquea el mutex del siguiente nodo (temp_p->next) para evitar cambios mientras se accede a él. Se imprime un mensaje indicando que el hilo ha bloqueado el nodo actual. Se desbloquea el mutex del nodo anterior (temp_p).
- Desbloqueo del nodo head: Se verifica si el nodo actual es el nodo head_p. Si es así, se desbloquea el mutex head_p_mutex y se imprime un mensaje indicando que el hilo ha desbloqueado el nodo head.
- Desbloqueo del mutex del nodo actual: Se desbloquea el mutex del nodo actual (temp_p).

6. Read-Write Locks

6.1. Implementación

```c
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>
#include "timer.h"

#define MAX_THREAD 4

struct list_node {
    int data;
    struct list_node * next;
};

long a = 0;

struct list_node * head_p = NULL;
pthread_rwlock_t rwlock;

int member(int value) {
    struct list_node * curr_p = head_p;
    while(curr_p != NULL && curr_p->data < value) curr_p = curr_p->next;
    if(curr_p == NULL || curr_p->data > value) return 0;
    else return 1;
}

int insert (int value)
{
    struct list_node * curr_p = *(&head_p);
    struct list_node * prev_p = NULL;
    struct list_node * temp_p;

    while(curr_p != NULL && curr_p->data < value) {
        prev_p = curr_p;
        curr_p = curr_p->next;
    }

    if(curr_p == NULL || curr_p->data > value) {
        temp_p = malloc(sizeof(struct list_node));
        temp_p->data = value;
        temp_p->next = curr_p;
```

```
41          if(prev_p == NULL) *(&head_p) = temp_p;
42          else prev_p->next = temp_p;
43          return 1;
44      }else return 0; //valor igual a uno de la lista
45  }
46
47
48  int delete(int value) {
49      struct list_node * curr_p = *(&head_p);
50      struct list_node * prev_p = NULL;
51
52      while(curr_p != NULL && curr_p->data < value) {
53          prev_p = curr_p;
54          curr_p = curr_p->next;
55      }
56
57      if(curr_p != NULL && curr_p->data == value) {
58          if(prev_p == NULL) {
59              *(&head_p) = curr_p -> next;
60              free(curr_p);
61          }else{
62              prev_p->next = curr_p->next;
63              free(curr_p);
64          }
65          return 1;
66      }else {
67          return 0;
68      }
69  }
70
71  void sprint() {
72      struct list_node * curr_p = head_p;
73      while(curr_p != NULL) {
74          printf("%d " , curr_p->data);
75          curr_p = curr_p->next;
76      }
77  }
78  void* Insert(void * value) {
79      int val = *((int * )value);
80      pthread_t my_rank = pthread_self();
81      pthread_rwlock_wrlock(&rwlock);
82      printf("Hilo %d Bloquea insert\n" , my_rank);
83      insert(val);
84      pthread_rwlock_unlock(&rwlock);
85      printf("Hilo %d Desbloquea insert\n" , my_rank);
86  }
87
```

```
88 void* Member(void * value) {
89     int val = *((int * )value);
90     pthread_t my_rank = pthread_self();
91     pthread_rwlock_rdlock(&rwlock);
92     printf("Hilo %d Bloquea ,menber\n" , my_rank);
93     insert(val);
94     pthread_rwlock_unlock(&rwlock);
95     printf("Hilo %d Desbloquea member\n" , my_rank);
96 }
97
98
99 void* Delete(void * value) {
100    int val = *((int * )value);
101    pthread_t my_rank = pthread_self();
102    pthread_rwlock_wrlock(&rwlock);
103    printf("Hilo %d Bloquea delete\n" , my_rank);
104    insert(val);
105    pthread_rwlock_unlock(&rwlock);
106    printf("Hilo %d Desbloquea delete\n" , my_rank);
107 }
```

6.2. Ejecución

```
D:\UNSA\Barriers\list>a
Hilo 1 Bloquea insert
Hilo 1 Desbloquea insert
Hilo 4 Bloquea insert
Hilo 4 Desbloquea insert
Hilo 3 Bloquea insert
Hilo 3 Desbloquea insert
Hilo 2 Bloquea insert
Hilo 2 Desbloquea insert
Hilo 5 Bloquea member
Hilo 5 Desbloquea member
Elapsed time = 0.000000e+000 seconds
2 5 6 9
```

6.3. Análisis

- **pthread_rwlock_wrlock(&rwlock):** Esta función adquiere un bloqueo de escritura en el **'read-write lock'**. Un bloqueo de escritura impide que cualquier otro hilo adquiera un bloqueo de escritura o lectura hasta que el bloqueo de escritura actual se libere. En otras palabras, garantiza que solo un hilo a la vez pueda realizar operaciones de escritura críticas , en nuestro código si un hilo entra a la funcion

insert adquiere este bloqueo , eso quiere decir , que bloquea a otros hilos que quieran hacer una lectura o escritura en la lista enlazada , en otras palabras mientras se realiza el insert , no se podra realizar ni la operación de delete ni la de member hasta que termine este bloqueo , lo mismo sucede en la función delete.

- **pthread_rwlock_rdlock(rwlock):** Esta función adquiere un bloqueo de lectura en el read-write lock". A diferencia del bloqueo de escritura, múltiples hilos pueden adquirir bloqueos de lectura simultáneamente. Sin embargo, no es posible adquirir un bloqueo de escritura mientras existan bloqueos de lectura activos. Esto permite que varios hilos realicen operaciones de lectura de manera concurrente, lo que puede ser beneficioso para mejorar el rendimiento , en nuestro codigo pdemos observar que la única función que realiza este bloqueo es en la función member , esto quiere decir que mdiferentes hilos pueden ejecutar la funcion member siempre y cuando tenga un bloqueo de lectura , sin embargo , mientras se realiza member , no se podra realizar ni insert , ni delete porque solo se puede realizar la lectura de manera concurrente , por lo que se producira un bloqueo para estas funciones **delete** y **member**.

7. Comparación del rendimiento de las implementaciones de Linked List

Cuadro 1: 100 000 ops , 90 % insert , 10 % member , 10 % delete

n° pthreads	Read-Write Locks	One Mutex for Entire List	One Mutex per Node
1	2.51	3.00	11.56
2	4.37	6.17	30.6
4	5.10	5.20	15.12
8	4.80	4.98	14.0

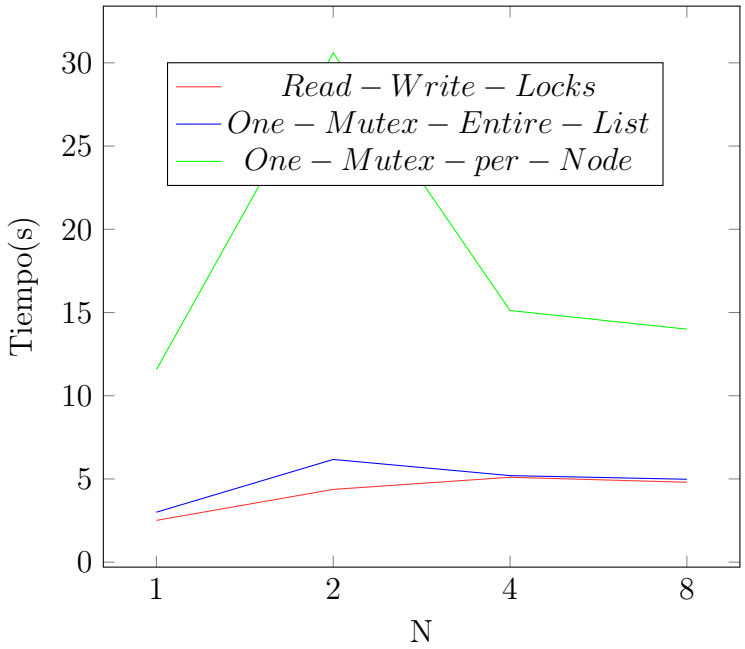


Figura 1: Tiempo de ejecución de las diferentes implementaciones

Cuadro 2: 100 000 ops , 90 % member , 10 % insert , 10 % delete

n° pthreads	Read-Write Locks	One Mutex for Entire List	One Mutex per Node
1	0.203	0.223	1.420
2	0.113	0.431	5.125
4	0.110	0.40	3.325
8	0.180	0.437	3.000

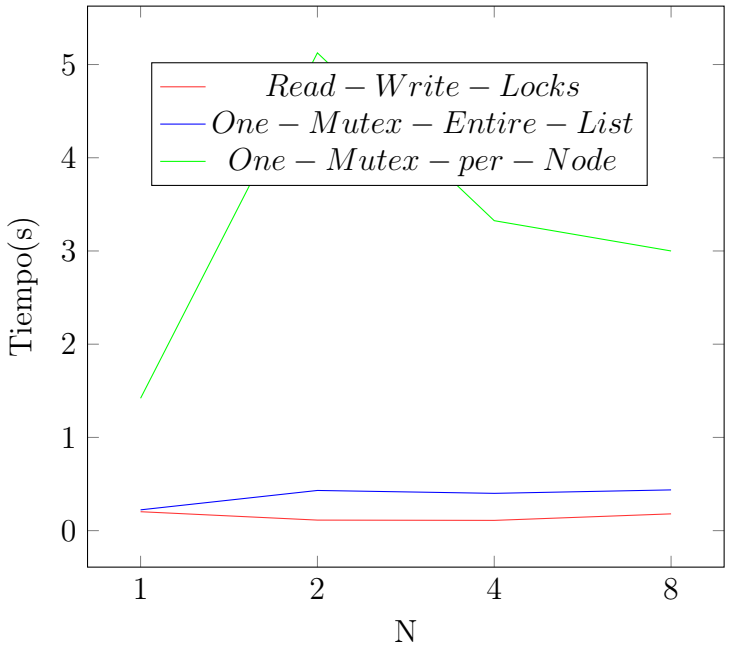


Figura 2: Tiempo de ejecución de las diferentes implementaciones

7.1. Conclusiones

- La implementación de read-write lockscz un solo mutex muestran un rendimiento similar, ya que ambas serializan las operaciones y no hay conflicto para el bloqueo. La sobrecarga asociada con ambas implementaciones consiste en una llamada a la función antes y después de las operaciones en la lista.
- En contraste, la implementación que utiliza un mutex por nodo es considerablemente más lenta. Esto se debe a que cada acceso a un nodo implica dos llamadas a funciones: una para bloquear el mutex del nodo y otra para desbloquearlo. Como resultado, esta implementación presenta una mayor sobrecarga.
- La desventaja de la implementación con un mutex por nodo persiste al utilizar múltiples hilos, ya que la sobrecarga de bloqueos y desbloqueos la hace menos competitiva en comparación con las otras dos.
- Una diferencia notable entre las tablas es el rendimiento relativo de las implementaciones de read-write lockz un solo mutex cuando se emplean varios hilos. En escenarios con pocas inserciones y eliminaciones, la implementación de read-write lock"supera a la de un solo mutex, sugiriendo que los read-write locks"permiten un acceso concurrente eficiente a la lista. No obstante, cuando hay un número considerable de inserciones y eliminaciones, la diferencia de rendimiento entre ambas implementaciones es mínima.
- Es importante destacar que, al usar un solo mutex o un mutex por nodo, el programa siempre es igual de rápido o más rápido cuando se ejecuta con un solo hilo. Además, cuando el número de inserciones y eliminaciones es considerable, la implementación con read-write locks"también es más rápida con un solo hilo. Esto no sorprende en

la implementación con un solo mutex, ya que los accesos a la lista están serializados. Sin embargo, con read-write locks", parece que la contención por los bloqueos de escritura afecta negativamente el rendimiento cuando son sustanciales.