Trabajo de Investigación sobre las 4 Barreras en la Programación en Memoria Compartida

Patrick Xavier Márquez Choque
Univesidad Católica San Pablo
Arequipa, Perú
Email: patrick.marquez@ucsp.edu.pe

Abstract Este trabajo tiene como objetivo investigar las 4 barreras en la programación en memoria compartida mencionadas en el capítulo 4 del libro de Peter Pacheco y realizar una comparación de las implementaciones mencionadas en el libro.

Todas las pruebas de ejecución de los programas implementados se utilizan con un Hardware Promedio con una CPU Intel (R) Pentium(R) CPU G450 @ 3.50GHz con 2 Núcleos y 4 Procesadores lógicos con 12.0GB de Memoria RAM.

1 Barreras en Pthreads

El link al repositorio donde se encuentra implementado los códigos mencionados se encuentra en el siguiente enlace: https://github.com/patrick03524/Programacion-Paralela-y-Distribuida/tree/main/SextoTrabajo/Barreras

1.1 Busy-Waiting y un Mutex

Esta barrera implica ya lo revisado en el libro del Busy-Waiting pero utilizando un mutex. El Busy-Waiting comparte la lógica similar a un cuello de botella en cuestión de núcleos que comparten un recurso en común que puede ser un contador y que se represente a través de todas las threads en orden utilizando el mutex de tal manera que se asegure que este recurso compartido no sea utilizado por más de una thread a la vez y así asegurar la fidelidad de este recursos. El problema reside ya que este punto dentro del programa llamado punto de sincronización donde cada thread esta asegurandose que todos estén en el mismo punto se llama la Barrera. Un ejemplo explicado en el libro de Peter Pacheco e implementado en c es el siguiente:

```
1 #include <stdio.h>
```


doi:10.1017/xxxxxxx xxxx

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

#define BARRIER_COUNT 100

int thread_count;
int barrier_thread_counts[BARRIER_COUNT];
pthread_mutex_t barrier_mutex;

void Usage(char* prog_name); /* Funtion that ask the number of pth */
void *Thread_work(void* rank); /* Work of every Thread */

int main(int argc, char* argv[]) {
   long       thread, i;
   pthread_t* thread_handles;
   double start, finish;
   if (argc != 2)
      Usage(argv[0]);
   thread_count = strtol(argv[1], NULL, 10);
   thread_handles = malloc (thread_count*sizeof(pthread_t));
   for (i = 0; i < BARRIER_COUNT; i++)
      barrier_thread_counts[i] = 0;
   pthread_mutex_init(&barrier_mutex, NULL);
   GET_TIME(start);
   for (thread = 0; thread < thread_count; thread++)
      pthread_create(&thread_handles[thread], NULL,
          Thread_work, (void*) thread);

   for (thread = 0; thread < thread_count; thread++) {
      pthread_join(thread_handles[thread], NULL);
   }
   GET_TIME(finish);
   /* GET TIME */
   printf("Elapsed time = %e seconds\n", finish - start);
   /* FREE PTHREADS */
   pthread_mutex_destroy(&barrier_mutex);
   free(thread_handles);
   return 0;
}  /* main */

void Usage(char* prog_name) {

   fprintf(stderr, "usage: %s <number of threads>\n", prog_name);
   exit(0);
}  /* Usage */

void *Thread_work(void* rank) {
   for (int i = 0; i < BARRIER_COUNT; i++) {
      pthread_mutex_lock(&barrier_mutex);
      barrier_thread_counts[i]++;
      pthread_mutex_unlock(&barrier_mutex);
      while (barrier_thread_counts[i] < thread_count);
   }
   return NULL;
```

```
57 }  /* Thread_work */
```

Listing 1: Barrera Busy-Waiting Mutex

Dentro del Listing 1 : Barrera Busy-Waiting Mutex es donde podemos observar el trabajo que se mencionó anteriormente ya que este código fue implementado del libro. Podemos concluir que debido a este lógica de cuello de botella su rendimiento puede ser no ideal cuando se utilicen muchos más threads de los núcleos físicos dentro del computador utilizado pero esto se verá en profundidad en el punto 5 Comparación de estas barreras.

1.2 Semáforos

La barrera del código implementado en el libro utilizará los semáforos dentro de la librería «Semaphore.h» con el objetivo de mejorar las deficiencias y los problemas del anterior problema pero la complejidad de este algoritmo se basa en que se tiene un contador que determinará cuantas threads han entrado dentro de la barrera usando esencialmente 2 semáforos como si se tratase de vías de conducción de una ciudad con doble vía de entrada donde la primera vez los semáforos serán desbloqueado, así la primer threads tendrá el paso abierto hasta que llegue a la barrera como un estado «semaphore wait», subsecuente mente hará su trabajo bloqueando el acceso del recurso compartida similar a una cola en caso si la thread utilice el recurso compartido.

Un ejemplo explicado en el libro de Peter Pacheco e implementado en c es el siguiente:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4  #include <semaphore.h>
5  #include "timer.h"
6
7  #define BARRIER_COUNT 100
8
9  int thread_count;
10 int counter;
11 sem_t barrier_sems[BARRIER_COUNT];
12 sem_t count_sem;
13
14 void Usage(char* prog_name); /* Funtion that ask the number of pth */
15 void *Thread_work(void* rank); /* Work of every Thread */
16
17 int main(int argc, char* argv[]) {
18    long       thread, i;
19    pthread_t* thread_handles;
20    double start, finish;
21    if (argc != 2)
22       Usage(argv[0]);
23    thread_count = strtol(argv[1], NULL, 10);
24    thread_handles = malloc (thread_count*sizeof(pthread_t));
25    for (i = 0; i < BARRIER_COUNT; i++)
26       sem_init(&barrier_sems[i], 0, 0);
```

```
27    sem_init(&count_sem, 0, 1);
28    GET_TIME(start);
29    for (thread = 0; thread < thread_count; thread++)
30       pthread_create(&thread_handles[thread], (pthread_attr_t*) NULL,
31           Thread_work, (void*) thread);
32
33    for (thread = 0; thread < thread_count; thread++) {
34       pthread_join(thread_handles[thread], NULL);
35    }
36    GET_TIME(finish);
37    /* GET TIME */
38    printf("Elapsed time = %e seconds\n", finish - start);
39    /* FREE PTHREADS */
40    sem_destroy(&count_sem);
41    for (i = 0; i < BARRIER_COUNT; i++)
42       sem_destroy(&barrier_sems[i]);
43    free(thread_handles);
44    return 0;
45 }  /* main */
46
47 void Usage(char* prog_name) {
48
49    fprintf(stderr, "usage: %s <number of threads>\n", prog_name);
50    exit(0);
51 }  /* Usage */
52
53 void *Thread_work(void* rank) {
54    for (int i = 0; i < BARRIER_COUNT; i++) {
55       sem_wait(&count_sem);
56       if (counter == thread_count - 1) {
57          counter = 0;
58          sem_post(&count_sem);
59          for (int j = 0; j < thread_count-1; j++)
60             sem_post(&barrier_sems[i]);
61       } else {
62          counter++;
63          sem_post(&count_sem);
64          sem_wait(&barrier_sems[i]);
65       }
66    }
67
68    return NULL;
69 }  /* Thread_work */
```

Listing 2: Barrera Semáforo

El código implementado utiliza una condicional fundamental dentro del semáforo ya que se va a comprobar si el contador es menor al número de threads restantes para que se pueda acceder en simultáneo a este recurso en caso sean pocas threads así incrementando dentro del bucle suponiendo que este trabajo tenga un mejor rendimiento que la Barrera Busy-Waiting con un Mutex.

1.3 Variables Condicionales

Esta es una manera algo curiosa de crear una barrera utilizando variables condicionales ya que esta condición se utilizará como una función o un objeto que permita a una thread para que se pueda suspender su ejecución hasta que esta condición se cumpla dentro de toda la compilación. Este evento permite que otras threads una señal que hará que se activen asociándose con un mutex.

Un ejemplo explicado en el libro de Peter Pacheco e implementado en c es el siguiente:

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

#define BARRIER_COUNT 100

int thread_count;
int barrier_thread_count = 0;
pthread_mutex_t barrier_mutex;
pthread_cond_t ok_to_proceed;

void Usage(char* prog_name); /* Funtion that ask the number of pth */
void *Thread_work(void* rank); /* Work of every Thread */

int main(int argc, char* argv[]) {
   long       thread;
   pthread_t* thread_handles;
   double start, finish;
   thread_count = strtol(argv[1], NULL, 10);
   thread_handles = malloc (thread_count*sizeof(pthread_t));
   pthread_mutex_init(&barrier_mutex, NULL);
   pthread_cond_init(&ok_to_proceed, NULL);
   GET_TIME(start);
   /* Initialize every pthread */
   for (thread = 0; thread < thread_count; thread++)
      pthread_create(&thread_handles[thread], NULL,
          Thread_work, (void*) thread);

   for (thread = 0; thread < thread_count; thread++) {
      pthread_join(thread_handles[thread], NULL);
   }
   /* Work Done*/
   GET_TIME(finish);
   /* GET TIME */
   printf("Elapsed time = %e seconds\n", finish - start);
   /* FREE PTHREADS */
   pthread_mutex_destroy(&barrier_mutex);
   pthread_cond_destroy(&ok_to_proceed);
   free(thread_handles);
   return 0;
}  /* main */

void Usage(char* prog_name) {
```

```
45
46    fprintf(stderr, "usage: %s <number of threads>\n", prog_name);
47    exit(0);
48 }  /* Usage */
49
50 void *Thread_work(void* rank) {
51    for (int i = 0; i < BARRIER_COUNT; i++) {
52       /* Barrier: 4.8.3 */
53       pthread_mutex_lock(&barrier_mutex);
54       barrier_thread_count++;
55       if (barrier_thread_count == thread_count) {
56          barrier_thread_count = 0;
57          pthread_cond_broadcast(&ok_to_proceed);
58       } else {
59          while (pthread_cond_wait(&ok_to_proceed,
60                    &barrier_mutex) != 0);
61       }
62       pthread_mutex_unlock(&barrier_mutex);
63    }
64    return NULL;
65 }  /* Thread_work */
```

Listing 3: Barrera Variable Condicional

El código implementado utilizará un mutex para asegurarse que el trabajo dentro de la barrera y se pueda cumplir esta condición correctamente en conjunto con todas las threads. Dentro de la tarea de cada thread se llamará a través de un handler de cada thread que desbloqueará el mutex para que los demás threads puedan acceder al recurso compartido. Dentro del código implementado se utilizará un broadcast para que se pueda acceder desde una thread a las demás en acceso del recurso y asi tenga un mejor rendimiento.

1.4 Barreras propias de la librería Pthreads

Dentro del libro de Peter Pacheco no se explica específicamente como es que trabaja esta barrera ya que es la que está implementada por defecto dentro de la propia librería nativa de «pthread.h» ya que se explica que un framework referencia-do tiene todo implementado pero se recurrió a una propia implementación utilizando las funciones de join de la librería de la siguiente manera:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4  #include <unistd.h>
5  #include "timer.h"
6
7  #define BARRIER_COUNT 100
8  #define PTHREADS_COUNT 64
9  int thread_count;
10 int barrier_pthread_count = 0;
11 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
12 pthread_cond_t barrier = PTHREAD_COND_INITIALIZER;
```

```
13
14 void Usage(char* prog_name); /* Funtion that ask the number of pth */
15 void *Thread_work(void* rank);  /* Work of every Thread*/
16 void *Main_Thread_work(void* rank);  /* Work of the Main Thread */
17
18 int main(int argc, char* argv[]) {
19   double start, finish;
20   pthread_t pthreads[PTHREADS_COUNT];
21     int p_id[PTHREADS_COUNT];
22     if (argc != 2)
23       Usage(argv[0]);
24     thread_count = strtol(argv[1], NULL, 10);
25     for(int i=0;i<thread_count;i++){
26       p_id[i]=i;
27     }
28     for(int i=1;i<thread_count;i++){
29       pthread_create(&pthreads[i], NULL, Thread_work, &p_id[i]);
30     }
31     sleep(1);
32     pthread_create(&pthreads[0], NULL, Main_Thread_work, &p_id[0]);
33     GET_TIME(start);
34     for(int i=0;i<thread_count;i++)
35   {
36      pthread_join(pthreads[i], NULL);
37      printf("pthread %i joined\n",i);
38   }
39   GET_TIME(finish);
40   /* GET TIME */
41    printf("Elapsed time = %e seconds\n", finish - start);
42    /* FREE PTHREADS */
43   pthread_mutex_destroy(&mutex);
44     pthread_cond_destroy(&barrier);
45   return 0;
46 }
47
48 void Usage(char* prog_name) {
49
50    fprintf(stderr, "usage: %s <number of threads>\n", prog_name);
51    exit(0);
52 }  /* Usage */
53
54 void *Thread_work(void* rank) {
55
56   int *id = (int *) rank;
57   printf("PThread %d waiting on the Main PThread waiting on mutex\n",*
       id);
58   pthread_mutex_lock(&mutex);
59   printf("PThread %d waiting on the Main PThread - mutex - waiting on
       condition\n",*id);
60   pthread_cond_wait(&barrier,&mutex);
61   printf("PThread %d execution - unlocking mutex\n",*id);
62   pthread_mutex_unlock(&mutex);
63   printf("PThread %d completing its work - mutex unlocked\n",*id);
64     return NULL;
65 }  /* Thread_work */
```

```
66
67 void *Main_Thread_work(void* rank) {
68
69   int *id = (int *) rank;
70   printf("Main PThread %d - letting the PThreads - waiting on mutex\n"
       ,*id);
71   pthread_mutex_lock(&mutex);
72   printf("Main PThread %d - letting the PThreads - got mutex -
       broadcast\n",*id);
73   pthread_cond_broadcast(&barrier);
74   printf("Main PThread %d - unlocking mutex\n",*id);
75   pthread_mutex_unlock(&mutex);
76   printf("Main PThread %d - work done - PThreads can go on now\n",*id)
       ;
77
78     return NULL;
79 }  /* Main Thread_work */
```

Listing 4: Barrera Librería Pthreads

Este algoritmo tiene la lógica similar al de las implementaciones anteriores pero con la diferencia que existirá una thread principal esencialmente donde esta será la primera en ser utilizada pero que necesita de un mutex para que se pueda bloquear del recurso compartido de los demás threads, ya que únicamente es prioridad que esta threads acabé su trabajo para que pueda esperar a repartir este recurso a las demás threads. Es la principal desventaja ya que el rendimiento depende esencialmente de esta thread ya que primero bloqueará el recurso con el mutex con la instrucción de «pthread cond wait». De cierta manera la ventaja será que cuando se cumpla el trabajo entonces será que las otras threads puedan acceder a su trabajo realizando un broadcast para las demás bloqueando y desbloqueando el mutex y esperando a cada thread.

1.5 Comparación de estas barreras

Se realizó una comparación de los tiempos de los algoritmos implementados de cada barrera en cuestión de tiempos que se puede apreciar de la siguiente manera:

Podemos concluir en primer lugar que:

- La primera barrera de Busy-Waiting y un Mutex es el algoritmo que más se demora ya que su rendimiento por su lógica tienen a empeorar muy significativamente cuando se agregan o crean threads de las que se tienen físicamente, entonces es aquí donde se puede apreciar las grandes cantidades de tiempo en comparación de los otros códigos,
- se probaron todos los algoritmos hasta la cantidad de 1024 threads exceptuando de la primera barrera ya que el tiempo excedía del deseado y es demasiado largo como para ser mostrado.
- La barrera del semáforo tiene la característica donde el método de seguridad evita que alguna threads se escape y no ocurra problemas, eso significa que

Busy-Waiting Núcleos	Time	Segundos
4	7.07E-04	0.00071
8	3.49E+00	3.49000
16	1.04E+01	10.40497
32	1.04E+01	24.50193
64	5.24E+01	52.37810
1024	-	-
Semaforós	Time	Segundos
4	2.07E-03	0.00207
8	4.08E-03	0.00408
16	7.13E-03	0.00713
32	1.30E-02	0.01300
64	2.82E-02	0.02820
1024	2.903819	
Variables Condicionales	Time	Segundos
4	3.04E-03	0.00304
8	3.52E-03	0.00352
16	5.32E-03	0.00532
32	1.02E-02	0.01020
64	1.79E-02	0.01790
1024	0.430871	
Pthread_Barrier	Time	Segundos
4	2.01E-02	0.02005
8	3.83E-02	0.03828
16	6.20E-02	0.06196
32	8.40E-02	0.08396
64	1.70E-01	0.17046
1024	1.626619	1.62662
4056	2.84E+00	2.84139

Figure 1: Cuadro comparativo de todas las Barreras

como esta encerrado dentro del mismo bloque de ejecución es lo que podemos apreciar que es la barrera con el algoritmo que tiene los tiempos más pequeños.

- Las variables condicionales tiene un método de seguridad para asegurarse que dentro de la barrera todos los threads estén en el mismo estado eso significa que gracias a esta condicional que reparte el recurso compartida será lo que represente una mayor cantidad de tiempo a comparación de la barrera que utiliza semáforos.

- Y en cambio en la última barrera nativa que se tiene en la librería pthread.h es la que se utiliza con respecto cuando se liberan cada una de las threads y es la que depende del proceso que se está utilizando, debido a que en la implementación solo esta esperando a la única thread principal para que termine y luego el mutex se liberé y acceder al recurso compartido es por eso que tenemos esa cantidad de tiempos significativamente bajos pero eso depende de la implementación de

cada thread y es por lo tanto que es el algoritmo mucho más flexible en cuestión de la implementación.

2 Lista Enlazada MultiThreads

El link al repositorio donde se encuentra implementado los códigos mencionados se encuentra en el siguiente enlace: https://github.com/patrick03524/Programacion-Paralela-y-Distribuida/tree/main/SextoTrabajo/ListaEnlazadaMu

Para la implementación de la Lista Enlazada en Multithreading se realizó la implementación mostrada en el libro de la siguiente manera:

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

struct list_node_s {
  int data;
  struct list_node_s* next;
  pthread_mutex_t mutex;
};

struct list_node_s* head = NULL;
int thread_count;
int total_ops;
double insert_p;
double search_p;
double delete_p;
int Member_count = 0, insert_count = 0, delete_count = 0;

/* List operations */
int Insert(int value);
void Print(void);
int Member(int value);
int Delete(int value);
void Free_list(void);
int Is_empty(void);

int Insert(int value) {
   struct list_node_s* curr = head;
   struct list_node_s* pred = NULL;
   struct list_node_s* temp;
   int rv = 1;
   while (curr != NULL && curr->data < value) {
      pred = curr;
      curr = curr->next;
   }
   if (curr == NULL || curr->data > value) {
      temp = malloc(sizeof(struct list_node_s));
      temp->data = value;
```

```
41          temp->next = curr;
42          if (pred == NULL)
43             head = temp;
44          else
45             pred->next = temp;
46       } else { /* value in list */
47          rv = 0;
48       }
49
50       return rv;
51    }  /* Insert */
52
53    void Print(void) {
54       struct list_node_s* temp;
55       printf("lista = ");
56       temp = head;
57       while (temp != (struct list_node_s*) NULL) {
58          printf("%d ", temp->data);
59          temp = temp->next;
60       }
61       printf("\n");
62    }  /* Print */
63
64    int  Member(int value) {
65       struct list_node_s* temp;
66       temp = head;
67       while (temp != NULL && temp->data < value)
68          temp = temp->next;
69       if (temp == NULL || temp->data > value) {
70          return 0;
71       } else {
72          return 1;
73       }
74    }  /* Member */
75
76    int Delete(int value) {
77       struct list_node_s* curr = head;
78       struct list_node_s* pred = NULL;
79       int rv = 1;
80       /* Member value */
81       while (curr != NULL && curr->data < value) {
82          pred = curr;
83          curr = curr->next;
84       }
85       if (curr != NULL && curr->data == value) {
86          if (pred == NULL) { /* first element in list */
87             head = curr->next;
88             free(curr);
89          } else {
90             pred->next = curr->next;
91             free(curr);
92          }
93       } else { /* Not in list */
94          rv = 0;
95       }
```

```
96
97    return rv;
98 }  /* Delete */
99
100 void Free_list(void) {
101    struct list_node_s* p;
102    struct list_node_s* temp;
103    if (Is_empty()) return;
104    p = head;
105    temp = p->next;
106    while (temp != NULL) {
107       free(p);
108       p = temp;
109       temp = p->next;
110    }
111    free(p);
112 }  /* Free_list */
113
114 int  Is_empty(void) {
115    if (head == NULL)
116       return 1;
117    else
118       return 0;
119 }  /* Is_empty */
120 }  /* Main Thread_work */
```

Listing 5: Barrera Librería Pthreads

Se implementaron las 4 funciones principales que son las de Insert, Member que es la función utilizada como find para encontrar los elementos dentro de la lista para asegurarse que la lista no tenga elementos repetidos, Delete para borrar elementos dentro de la lista y Free List como función para limpiar toda la lista enlazada.

Pero según la implementación de la lista enlazada tenemos en cuenta 3 diferentes implementaciones de la siguiente manera:

2.1 Implementación Pthreads Read-Write Locks

Esta implementación utiliza la lógica Red-Write Locks

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "timer.h"
4  #include "rand.h"
5  #include <pthread.h>
6
7  const int MAX_KEY = 100000000;
8  int act_keys = 100;
9
10 struct list_node_s {
11   int data;
12   struct list_node_s* next;
13   pthread_mutex_t mutex;
14 };
```

```
15
16 struct list_node_s* head = NULL;
17 int thread_count;
18 int total_ops;
19 double insert_p;
20 double search_p;
21 double delete_p;
22 pthread_rwlock_t rwlock;
23 pthread_mutex_t count_mutex;
24 int Member_count = 0, insert_count = 0, delete_count = 0;
25
26 /* Setup */
27 void Usage(char* prog_name);
28 void Get_input(int* inserts_in_main_p);
29
30 /* Thread function */
31 void* Thread_work(void* rank);
32
33 /* List operations */
34 int Insert(int value);
35 void Print(void);
36 int Member(int value);
37 int Delete(int value);
38 void Free_list(void);
39 int Is_empty(void);
40
41 int main(int argc, char* argv[]) {
42    long i;
43    int key, success, attempts;
44    pthread_t* thread_handles;
45    int inserts_in_main;
46    unsigned seed = 1;
47    double start, finish;
48    if (argc != 2) Usage(argv[0]);
49    thread_count = strtol(argv[1],NULL,10);
50    Get_input(&inserts_in_main);
51    i = attempts = 0;
52    while ( i < inserts_in_main && attempts < 2*inserts_in_main ) {
53       key = my_rand(&seed) % MAX_KEY;
54       success = Insert(key);
55       attempts++;
56       if (success) i++;
57    }
58    printf("Inserted %ld keys in empty list\n", i);
59    thread_handles = malloc(thread_count*sizeof(pthread_t));
60    pthread_mutex_init(&count_mutex, NULL);
61    pthread_rwlock_init(&rwlock, NULL);
62    GET_TIME(start);
63    for (i = 0; i < thread_count; i++)
64       pthread_create(&thread_handles[i], NULL, Thread_work, (void*) i)
       ;
65
66    for (i = 0; i < thread_count; i++)
67       pthread_join(thread_handles[i], NULL);
68    GET_TIME(finish);
```

```
69    Print();
70    printf("Elapsed time = %e seconds\n", finish - start);
71    printf("Total ops = %d\n", total_ops);
72    printf("Member(FIND) ops = %d\n", Member_count);
73    printf("insert ops = %d\n", insert_count);
74    printf("delete ops = %d\n", delete_count);
75    /* Free List and Threads */
76    Free_list();
77    pthread_rwlock_destroy(&rwlock);
78    pthread_mutex_destroy(&count_mutex);
79    free(thread_handles);
80    return 0;
81 }  /* main */
82
83 void Usage(char* prog_name) {
84    fprintf(stderr, "usage: %s <thread_count>\n", prog_name);
85    exit(0);
86 }  /* Usage */
87
88 void Get_input(int* inserts_in_main_p) {
89    printf("Keys: \n");
90    scanf("%d", inserts_in_main_p);
91    printf("Ops Number: \n");
92    scanf("%d", &total_ops);
93    printf("Percent of Ops in searches? (between 0 and 1)\n");
94    scanf("%lf", &search_p);
95    printf("Percent of Ops in inserts? (between 0 and 1)\n");
96    scanf("%lf", &insert_p);
97    delete_p = 1.0 - (search_p + insert_p);
98 }  /* Get_input */
99
100 void* Thread_work(void* rank) {
101    long my_rank = (long) rank;
102    int i, val;
103    double which_op;
104    unsigned seed = my_rank + 1;
105    int my_Member_count = 0, my_insert_count=0, my_delete_count=0;
106    int ops_per_thread = total_ops/thread_count;
107    for (i = 0; i < ops_per_thread; i++) {
108       which_op = my_drand(&seed);
109       val = my_rand(&seed) % MAX_KEY;
110       if (which_op < search_p) {
111          pthread_rwlock_rdlock(&rwlock);
112          Member(val);
113          pthread_rwlock_unlock(&rwlock);
114          my_Member_count++;
115       } else if (which_op < search_p + insert_p) {
116          pthread_rwlock_wrlock(&rwlock);
117          Insert(val);
118          pthread_rwlock_unlock(&rwlock);
119          my_insert_count++;
120       } else { /* delete */
121          pthread_rwlock_wrlock(&rwlock);
122          Delete(val);
123          pthread_rwlock_unlock(&rwlock);
```

```
124             my_delete_count++;
125          }
126       }  /* for */
127       pthread_mutex_lock(&count_mutex);
128       Member_count += my_Member_count;
129       insert_count += my_insert_count;
130       delete_count += my_delete_count;
131       pthread_mutex_unlock(&count_mutex);
132       return NULL;
133    }  /* Thread_work */
```

Listing 6: Barrera Librería Pthreads

2.2 Implementación Pthreads One Mutex for the Entire List

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4  #include "rand.h"
5  #include "timer.h"
6
7  const int MAX_KEY = 100000000;
8  int act_keys = 100;
9
10 struct list_node_s {
11    int data;
12    struct list_node_s* next;
13    pthread_mutex_t mutex;
14 };
15
16 struct list_node_s* head = NULL;
17 int thread_count;
18 int total_ops;
19 double insert_p;
20 double search_p;
21 double delete_p;
22 pthread_mutex_t mutex;
23 pthread_mutex_t count_mutex;
24 int member_total=0, insert_total=0, delete_total=0;
25
26 /* Setup */
27 void Usage(char* prog_name);
28 void Get_input(int* inserts_in_main_p);
29
30 /* Thread function */
31 void* Thread_work(void* rank);
32
33 int main(int argc, char* argv[]) {
34    long i;
35    int key, success, attempts;
36    pthread_t* thread_handles;
37    int inserts_in_main;
38    unsigned seed = 1;
```

```
39    double start, finish;
40    if (argc != 2) Usage(argv[0]);
41    thread_count = strtol(argv[1],NULL,10);
42    Get_input(&inserts_in_main);
43    i = attempts = 0;
44    while ( i < inserts_in_main && attempts < 2*inserts_in_main ) {
45       key = my_rand(&seed) % MAX_KEY;
46       success = Insert(key);
47       attempts++;
48       if (success) i++;
49    }
50    printf("Inserted %ld keys in empty list\n", i);
51    thread_handles = malloc(thread_count*sizeof(pthread_t));
52    pthread_mutex_init(&mutex, NULL);
53    pthread_mutex_init(&count_mutex, NULL);
54    GET_TIME(start);
55    for (i = 0; i < thread_count; i++)
56       pthread_create(&thread_handles[i], NULL, Thread_work, (void*) i)
       ;
57
58    for (i = 0; i < thread_count; i++)
59       pthread_join(thread_handles[i], NULL);
60    GET_TIME(finish);
61    Print();
62    printf("Elapsed time = %e seconds\n", finish - start);
63    printf("Total ops = %d\n", total_ops);
64    printf("member ops = %d\n", member_total);
65    printf("insert ops = %d\n", insert_total);
66    printf("delete ops = %d\n", delete_total);
67    /* Free List and Threads */
68    Free_list();
69    pthread_mutex_destroy(&mutex);
70    pthread_mutex_destroy(&count_mutex);
71    free(thread_handles);
72    return 0;
73 }  /* main */
74
75 void Usage(char* prog_name) {
76    fprintf(stderr, "usage: %s <thread_count>\n", prog_name);
77    exit(0);
78 }  /* Usage */
79
80 void Get_input(int* inserts_in_main_p) {
81    printf("Keys: \n");
82    scanf("%d", inserts_in_main_p);
83    printf("Ops Number: \n");
84    scanf("%d", &total_ops);
85    printf("Percent of Ops in searches? (between 0 and 1)\n");
86    scanf("%lf", &search_p);
87    printf("Percent of Ops in inserts? (between 0 and 1)\n");
88    scanf("%lf", &insert_p);
89    delete_p = 1.0 - (search_p + insert_p);
90 }  /* Get_input */
91
92 void* Thread_work(void* rank) {
```

```
93    long my_rank = (long) rank;
94    int i, val;
95    double which_op;
96    unsigned seed = my_rank + 1;
97    int my_member=0, my_insert=0, my_delete=0;
98    int ops_per_thread = total_ops/thread_count;
99
100   for (i = 0; i < ops_per_thread; i++) {
101      which_op = my_drand(&seed);
102      val = my_rand(&seed) % MAX_KEY;
103      if (which_op < search_p) {
104         pthread_mutex_lock(&mutex);
105         Member(val);
106         pthread_mutex_unlock(&mutex);
107         my_member++;
108      } else if (which_op < search_p + insert_p) {
109         pthread_mutex_lock(&mutex);
110         Insert(val);
111         pthread_mutex_unlock(&mutex);
112         my_insert++;
113      } else { /* delete */
114         pthread_mutex_lock(&mutex);
115         Delete(val);
116         pthread_mutex_unlock(&mutex);
117         my_delete++;
118      }
119   }  /* for */
120   pthread_mutex_lock(&count_mutex);
121   member_total += my_member;
122   insert_total += my_insert;
123   delete_total += my_delete;
124   pthread_mutex_unlock(&count_mutex);
125   return NULL;
126}  /* Thread_work */
```

Listing 7: Barrera Librería Pthreads

2.3 Implementación Pthreads One Mutex per Node

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4  #include "rand.h"
5  #include "timer.h"
6
7  const int MAX_KEY = 100000000;
8  int act_keys = 100;
9
10 const int IN_LIST = 1;
11 const int EMPTY_LIST = -1;
12 const int END_OF_LIST = 0;
13
14 struct list_node_s {
```

```
15    int    data;
16    pthread_mutex_t mutex;
17    struct list_node_s* next;
18 };
19
20 struct list_node_s* head = NULL;
21 pthread_mutex_t head_mutex;
22 int thread_count;
23 int total_ops;
24 double insert_p;
25 double search_p;
26 double delete_p;
27 pthread_mutex_t count_mutex;
28 int member_total=0, insert_total=0, delete_total=0;
29
30 /* Setup */
31 void Usage(char* prog_name);
32 void Get_input(int* inserts_in_main_p);
33
34 /* Thread function */
35 void* Thread_work(void* rank);
36
37 int main(int argc, char* argv[]) {
38    long i;
39    int key, success, attempts;
40    pthread_t* thread_handles;
41    int inserts_in_main;
42    unsigned seed = 1;
43    double start, finish;
44    if (argc != 2) Usage(argv[0]);
45    thread_count = strtol(argv[1], NULL, 10);
46    Get_input(&inserts_in_main);
47    i = attempts = 0;
48    pthread_mutex_init(&head_mutex, NULL);
49    while ( i < inserts_in_main && attempts < 2*inserts_in_main ) {
50       key = my_rand(&seed) % MAX_KEY;
51       success = Insert(key);
52       attempts++;
53       if (success) i++;
54    }
55    printf("Inserted %ld keys in empty list\n", i);
56    thread_handles = malloc(thread_count*sizeof(pthread_t));
57    pthread_mutex_init(&count_mutex, NULL);
58
59    GET_TIME(start);
60    for (i = 0; i < thread_count; i++)
61       pthread_create(&thread_handles[i], NULL, Thread_work, (void*) i)
       ;
62
63    for (i = 0; i < thread_count; i++)
64       pthread_join(thread_handles[i], NULL);
65    GET_TIME(finish);
66    Print();
67    printf("Elapsed time = %e seconds\n", finish - start);
68    printf("Total ops = %d\n", total_ops);
```

```
69    printf("member ops = %d\n", member_total);
70    printf("insert ops = %d\n", insert_total);
71    printf("delete ops = %d\n", delete_total);
72    Free_list();
73    pthread_mutex_destroy(&head_mutex);
74    pthread_mutex_destroy(&count_mutex);
75    free(thread_handles);
76    return 0;
77 }  /* main */
78
79 void Usage(char* prog_name) {
80    fprintf(stderr, "usage: %s <thread_count>\n", prog_name);
81    exit(0);
82 }  /* Usage */
83
84 void Get_input(int* inserts_in_main_p) {
85    printf("Keys:\n");
86    scanf("%d", inserts_in_main_p);
87    printf("Ops Number:\n");
88    scanf("%d", &total_ops);
89    printf("Percent of Ops in searches? (between 0 and 1)\n");
90    scanf("%lf", &search_p);
91    printf("Percent of Ops in inserts? (between 0 and 1)\n");
92    scanf("%lf", &insert_p);
93    delete_p = 1.0 - (search_p + insert_p);
94 }  /* Get_input */
95
96 void Init_ptrs(struct list_node_s** curr_pp, struct list_node_s**
       pred_pp) {
97    *pred_pp = NULL;
98    pthread_mutex_lock(&head_mutex);
99    *curr_pp = head;
100   if (*curr_pp != NULL)
101      pthread_mutex_lock(&((*curr_pp)->mutex));
102 }  /* Init_ptrs */
103
104 void* Thread_work(void* rank) {
105    long my_rank = (long) rank;
106    int i, val;
107    double which_op;
108    unsigned seed = my_rank + 1;
109    int my_member=0, my_insert=0, my_delete=0;
110    int ops_per_thread = total_ops/thread_count;
111
112    for (i = 0; i < ops_per_thread; i++) {
113       which_op = my_drand(&seed);
114       val = my_rand(&seed) % MAX_KEY;
115       if (which_op < search_p) {
116          Member(val);
117          my_member++;
118       } else if (which_op < search_p + insert_p) {
119          Insert(val);
120          my_insert++;
121       } else { /* delete */
122          Delete(val);
```

```
123             my_delete++;
124         }
125     }  /* for */
126     pthread_mutex_lock(&count_mutex);
127     member_total += my_member;
128     insert_total += my_insert;
129     delete_total += my_delete;
130     pthread_mutex_unlock(&count_mutex);
131     return NULL;
132 }  /* Thread_work */
```

Listing 8: Barrera Librería Pthreads

2.4 Pruebas de las implementaciones

Toda la implementación de los códigos utilizando el hardware especificado en la parte inicial utilizando lo podemos apreciar en la siguiente imagen:

Implementation	Secuencia	1	2	4	8	64
Read-Write Locks	Cant Ops	100000	100000	100000	100000	100000
	Time	0.4048	0.2312	0.2313	0.2367	0.3554
One Mutex per List	Cant Ops	100000	100000	100000	100000	100000
	Time	0.3920	0.5857	0.6985	0.7239	1.1317
One Mutex per Node	Cant Ops	100000	100000	100000	100000	100000
	Time	5.2074	5.6999	3.5494	3.7982	4.6963

Figure 2: Cuadro comparativo de todas las Barreras

- En primer lugar las pruebas fueron tratadas de tal manera como en el libro, especificamente la primera prueba donde se tiene una cantidad de 1000 elementos que se van a insertar con 100000 operaciones en total con una cantidad del 99% operaciones de búsqueda, un 0.5% de operaciones de insertado y por lo tanto un 0.5% de operaciones de borrado utilizando números aleatorios para estas pruebas.

- Podemos observar la gran cantidad de tiempo que toma en comparación del algoritmo implementado cuando se tiene un mutex en cada uno de los nodos de tal manera que es muy restringida la operación de inserción volviéndolo casi secuencial y eso lo podemos apreciar en los tiempos que se tienen.

- Podemos apreciar la gran cantidad de similitudes si se tienen una mayor cantidad de pthreads pero es aquí que radica la diferencia ya que que depende de la arquitectura que podemos apreciar es como que las pruebas que utilizan 4 threads son la que tienen mejor rendimiento ya que estas son las que se pueden apreciar dentro de nuestra hardware.

3 Thread-Safety: Analizando StrTok

la función de Stktok es un método que cuando se utiliza en la llamada entonces ubicará o creará un puntero a un tipo de dato string de tal maenra que en subsecuentes llamadas de esta funciones retornará tokens sucesivamente a través de copias inicializando y que puede utilizar por ejemplo un semáforo para cada una de las threads que se le asigna en primer lugar, es debido esencialmente al tipo de recurso compartida que rompe un poco la lógica de thread safety ya que puede repartir este recurso a través del puntero que, aunque no sea lo mismo que compartirlo como tal esta permitiendo el acceso de este recurso de una cierta manera.