

# Laboratorio 03

PThreads

Autor: Juan Manuel Soto Begazo  
CUI: 20192267  
Docente: Dr. Alvaro Henry Mamani Aliaga  
Computación Paralela y Distribuida  
Grupo A  
Arequipa, Peru

# 1. Actividades

Las tareas a realizar son las siguientes:

- La implementación, ejecución y análisis de cada uno de los métodos de barrera mostrados en el libro (sección 4.8).
- La implementación, ejecución y análisis de la lista enlazada multithreading, usando cada una de las formas de implementación de la lista: Read-Write Locks; One Mutex for Entire List; y One Mutex per Node.

La implementación se encuentra en el siguiente repositorio de Git Hub.

## 2. Desarrollo

### 2.1. Metodos de Barrera

#### 2.1.1. Busy-waiting and a mutex

- Implementación

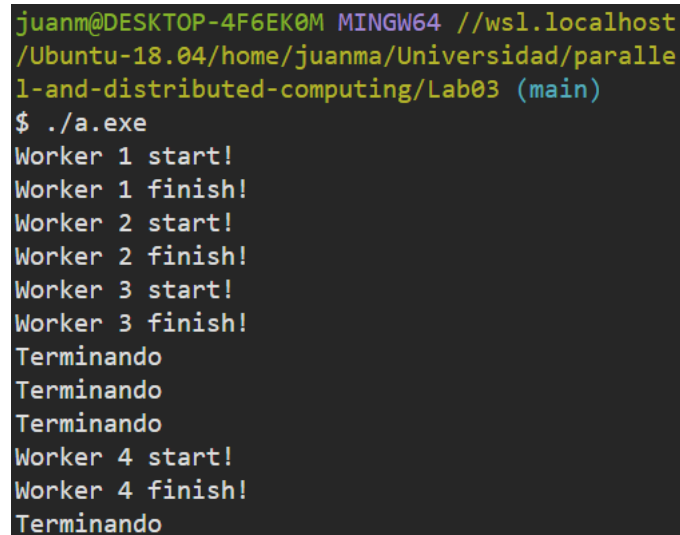
---

```
1  #include <iostream>
2  #include <pthread.h>
3  #define MAX_THREAD 4
4  using namespace std;
5
6  int counter = 0;
7  int thread_count = 4;
8  pthread_mutex_t barrier_mutex;
9  int do_nothing = 0;
10
11 void* Thread_work(void* arg) {
12     pthread_mutex_lock(&barrier_mutex);
13     counter++;
14
15     cout << "Worker " << counter << " start!\n";
16     for(int i = 0; i < 1e4; i++) do_nothing++;
17     std::cout << "Worker " << counter << " finish!\n";
18
19     pthread_mutex_unlock(&barrier_mutex);
20     while (counter < thread_count); // Esperando al ultimo hilo
21     cout << "Terminando\n";
22 }
23
24 int main(){
25     pthread_t threads[MAX_THREAD];
26
27     for (int i = 0; i < MAX_THREAD; i++)
```

```
28     pthread_create(&threads[i], NULL, Thread_work, (void*)NULL);
29
30     for (int i = 0; i < MAX_THREAD; i++)
31         pthread_join(threads[i], NULL);
32     return 0;
33 }
```

---

#### ■ Ejecución



```
juanm@DESKTOP-4F6EK0M MINGW64 //wsl.localhost
/Ubuntu-18.04/home/juanma/Universidad/paralel
1-and-distributed-computing/Lab03 (main)
$ ./a.exe
Worker 1 start!
Worker 1 finish!
Worker 2 start!
Worker 2 finish!
Worker 3 start!
Worker 3 finish!
Terminando
Terminando
Terminando
Worker 4 start!
Worker 4 finish!
Terminando
```

Figura 2.1: Ejecución Busy-waiting and a mutex

#### ■ Análisis

1. En la función *main*:

- Se crean 4 hilos en el primer bucle `for`.
- En el segundo bucle, se espera a que los hilos terminen.

2. En la función *Thread\_Work*:

- La función `pthread_mutex_lock(barrier_mutex)`; se encarga de bloquear el acceso a la sección crítica mediante la barrera de mutex.
- Dentro de la sección crítica, se incrementa el contador y se realiza “el trabajo crítico”. Al terminar, el *worker* anuncia que ha terminado.

3. Sección crítica:

- La sección crítica se libera para que los siguientes hilos puedan entrar.
- Mientras tanto, el hilo actual espera en el bucle de *Busy-waiting* hasta que el último hilo entre a la sección crítica.
- Cuando el cuarto hilo entra a la sección crítica y el contador aumenta, en ese preciso momento, todos los hilos que estuvieron esperando tendrán la libertad para salir del bucle de *Busy-waiting* y anunciarán que están “Terminando”.

4. Finalmente, en la función *main*:

- Se recibirá la señal de que todos los *threads* terminaron y, finalmente, el programa terminará.

De esta forma estamos asegurando que todos los hilos puedan acceder a la sección crítica en orden y además sincronizando los hilos para que cuando todos los hilos hayan finalizado su trabajo en la sección crítica podamos continuar realizando el resto del trabajo con la información actualizada por todos los hilos. Por ejemplo si los procesos están actualizando algún tipo de puntaje, esperaremos a que el puntaje haya sido actualizado por todos los hilos para continuar con el resto del trabajo.

### 2.1.2. Semaphores

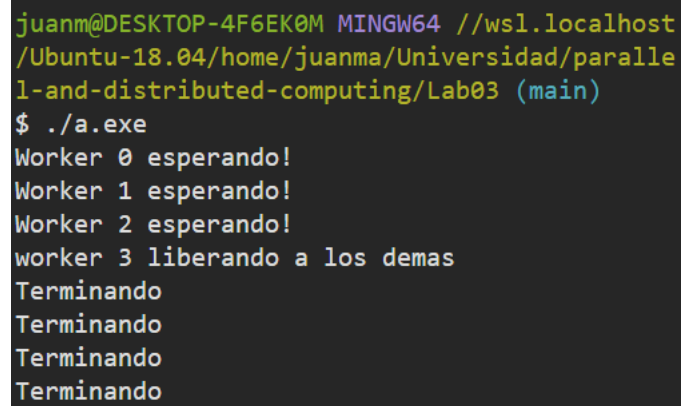
#### ■ Implementación

---

```
1  #include <iostream>
2  #include <pthread.h>
3  #include <semaphore.h>
4  #define MAX_THREAD 4
5  using namespace std;
6
7  int counter = 0;
8  int thread_count = 4;
9  sem_t count_sem;
10 sem_t barrier_sem;
11
12 void* Thread_work(void* arg){
13     sem_wait(&count_sem);
14     if (counter == thread_count - 1) {
15         cout << "worker " << counter << " liberando a los demas\n";
16         counter = 0;
17         sem_post(&count_sem);
18         for (int j = 0; j < thread_count - 1; j++) sem_post(&barrier_sem);
19     } else {
20         cout << "Worker " << counter << " esperando!\n";
21         counter++;
22         sem_post(&count_sem);
23         sem_wait(&barrier_sem);
24     }
25     cout << "Terminando\n";
26 }
27
28 int main(){
29     pthread_t threads[MAX_THREAD];
30     sem_init(&count_sem, 0, 1);
31     sem_init(&barrier_sem, 0, 0);
32 }
```

```
33     for (int i = 0; i < MAX_THREAD; i++)
34         pthread_create(&threads[i], NULL, Thread_work, (void*)NULL);
35
36     for (int i = 0; i < MAX_THREAD; i++)
37         pthread_join(threads[i], NULL);
38     return 0;
39 }
```

#### ■ Ejecución



```
juanm@DESKTOP-4F6EK0M MINGW64 //wsl.localhost
/Ubuntu-18.04/home/juanma/Universidad/paralle
l-and-distributed-computing/Lab03 (main)
$ ./a.exe
Worker 0 esperando!
Worker 1 esperando!
Worker 2 esperando!
worker 3 liberando a los demas
Terminando
Terminando
Terminando
Terminando
```

Figura 2.2: Ejecución Semaphores

#### ■ Análisis

1. Inicialización de Semáforos:
  - Se crean e inicializan dos semáforos: `count_sem` (un mutex) con un valor de 1 y `barrier_sem` (una barrera) con un valor de 0.
2. Creación de Hilos:
  - En la función `main`, se crean 4 hilos en un bucle utilizando `pthread_create`.
3. Ejecución de Hilos:
  - Cada hilo ejecuta la función `Thread_work`.
  - Los hilos entran en una sección crítica controlada por el semáforo `count_sem`. Este semáforo garantiza la exclusión mutua en el acceso al contador compartido.
4. Contador y Barrera:
  - El contador se utiliza para realizar un seguimiento de cuántos hilos han llegado a la sección crítica.
  - Si un hilo es el último en llegar a la sección crítica, reinicia el contador, libera a los demás hilos y anuncia que está liberando.
  - Si no es el último hilo, incrementa el contador, libera el semáforo `count_sem` y espera en el semáforo `barrier_sem`.
5. Sincronización y Finalización:

- Todos los hilos anuncian "Terminando" después de salir de la barrera.
- En la función `main`, se espera a que todos los hilos terminen utilizando `pthread_join`.

### 2.1.3. Condition variables

#### ■ Implementación

---

```
1  #include <iostream>
2  #include <pthread.h>
3  #include <semaphore.h>
4  #define MAX_THREAD 4
5  using namespace std;
6
7  int counter = 0;
8  int thread_count = 4;
9  pthread_mutex_t mutex;
10 pthread_cond_t cond_var;
11
12
13 void* Thread_work(void* arg) {
14     pthread_mutex_lock(&mutex);
15     counter++;
16     cout << "Worker " << counter << " start\n";
17
18     if (counter == thread_count) {
19         cout << "Ultimo worker liberando a los demas\n";
20         counter = 0;
21         pthread_cond_broadcast(&cond_var);
22     } else {
23         while (pthread_cond_wait(&cond_var, &mutex) != 0);
24     }
25     pthread_mutex_unlock(&mutex);
26     cout << "Terminando\n";
27     return NULL;
28 }
29
30 int main(){
31     pthread_t threads[MAX_THREAD];
32     pthread_mutex_init(&mutex, NULL);
33     pthread_cond_init(&cond_var, NULL);
34     for (int i = 0; i < MAX_THREAD; i++)
35         pthread_create(&threads[i], NULL, Thread_work, (void*)NULL);
36
37     for (int i = 0; i < MAX_THREAD; i++)
38         pthread_join(threads[i], NULL);
39     return 0;
40 }
```

---

### ■ Ejecución

```
juanm@DESKTOP-4F6EK0M MINGW64 //wsl.localhost
/Ubuntu-18.04/home/juanma/Universidad/paralle
1-and-distributed-computing/Lab03 (main)
$ ./a.exe
Worker 1 start
Worker 2 start
Worker 3 start
Worker 4 start
Ultimo worker liberando a los demas
Terminando
Terminando
Terminando
Terminando
```

Figura 2.3: Ejecución Condition Variables

### ■ Análisis

1. Inicialización de Mutex y Variable de Condición:
  - Se inicializan un mutex (`mutex`) y una variable de condición (`cond_var`).
2. Creación de Hilos:
  - En la función `main`, se crean 4 hilos en un bucle utilizando `pthread_create`.
3. Ejecución de Hilos (`Thread_work`):
  - Cada hilo ejecuta la función `Thread_work`.
  - Se bloquea el acceso a la sección crítica mediante la adquisición del mutex.
  - Se incrementa el contador y se muestra un mensaje indicando que el trabajador ha comenzado.
4. Barrera con Variable de Condición:
  - Si el hilo actual es el último en llegar, reinicia el contador, muestra un mensaje indicando que el último trabajador está liberando a los demás y utiliza `pthread_cond_broadcast` para despertar a todos los hilos bloqueados.
  - Si no es el último hilo, entra en un bucle `while` y espera en la variable de condición hasta que sea despertado por el último hilo.
5. Fin de la Sección Crítica:
  - Después de salir de la barrera, se libera el mutex.
  - Se muestra un mensaje indicando que el trabajador ha terminado.
6. Espera a la Finalización de los Hilos:
  - En la función `main`, se espera a que todos los hilos terminen utilizando `pthread_join`.

## 2.2. Lista enlazada multithreading

### 2.2.1. Read-Write Locks

- Implementación

---

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <assert.h>
4  #include <pthread.h>
5  #include <unistd.h>
6
7  struct list_node_s{
8      int data;
9      struct list_node_s *next;
10 };
11
12 struct list_node_s *head_p;
13 pthread_rwlock_t    rwlock;
14
15 int member(int val){
16     struct list_node_s * curr_p = head_p;
17     while(curr_p != NULL && curr_p->data < val) curr_p = curr_p->next;
18     if(curr_p == NULL || curr_p->data > val) return 0;
19     else return 1;
20 }
21
22 int insert(int val) {
23     struct list_node_s* curr_p = head_p;
24     struct list_node_s* pred_p;
25     struct list_node_s* temp_p;
26     while(curr_p != NULL && curr_p->data < val){
27         pred_p = curr_p;
28         curr_p = curr_p->next;
29     }
30     if(curr_p == NULL || curr_p->data > val){
31         temp_p = malloc(sizeof (struct list_node_s));
32         temp_p->data = val;
33         temp_p->next = curr_p;
34         if(pred_p == NULL)
35             head_p = temp_p;
36         else
37             pred_p->next = temp_p;
38         return 1;
39     }
40     else{
41         return 0;
42     }
43 }
```



```
42 }
```

---

#### ■ Análisis

- `struct list_node_s`:
  - Estructura que representa un nodo en la lista enlazada.
  - Contiene un campo para almacenar datos (`data`) y un puntero al siguiente nodo (`next`).

## Variables Globales

- `struct list_node_s *head_p`:
  - Puntero a la cabeza de la lista enlazada.
- `pthread_rwlock_t rwlock`:
  - Candado de lectura-escritura utilizado para garantizar la sincronización al acceder y modificar la lista.

## Funciones

- `int member(int val)`:
  - Comprueba la membresía de un valor (`val`) en la lista enlazada.
  - Utiliza el candado de lectura-escritura para garantizar la consistencia durante la lectura.
  - Retorna 1 si el valor está presente en la lista, 0 en caso contrario.
- `int insert(int val)`:
  - Inserta un nuevo valor (`val`) en la lista enlazada de manera ordenada.
  - Utiliza el candado de lectura-escritura para garantizar la exclusión mutua durante la modificación.
  - Retorna 1 si la inserción es exitosa, 0 si el valor ya está presente en la lista.

### 2.2.2. One Mutex for Entire List

#### ■ Implementación

---

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4  struct list_node_s {
5      int    data;
6      pthread_mutex_t mutex;
7      struct list_node_s* next;
8  };
9  struct list_node_s* head = NULL;
```

```
10 pthread_mutex_t head_mutex;
11
12 int member(int value) {
13     struct list_node_s *temp, *old_temp;
14     pthread_mutex_lock(&head_mutex);
15     temp = head;
16     if (temp != NULL) pthread_mutex_lock(&(temp->mutex));
17     pthread_mutex_unlock(&head_mutex);
18     while(temp != NULL && temp->data < value){
19         if (temp->next != NULL)
20             pthread_mutex_lock(&(temp->next->mutex));
21         old_temp = temp;
22         temp = temp->next;
23         pthread_mutex_unlock(&(old_temp->mutex));
24     }
25
26     if(temp == NULL || temp->data > value){
27         if (temp != NULL)
28             pthread_mutex_unlock(&(temp->mutex));
29         return 0;
30     }
31     else {
32         pthread_mutex_unlock(&(temp->mutex));
33         return 1;
34     }
35 }
36
37 int Advance_ptrs(struct list_node_s curr_pp, struct list_node_s pred_pp) {
38     int rv = 1;
39     struct list_node_s* curr_p = *curr_pp;
40     struct list_node_s* pred_p = *pred_pp;
41
42     if (curr_p == NULL) {
43         if (pred_p == NULL) {
44             pthread_mutex_unlock(&head_mutex);
45             return -1;
46         } else {
47             return 0;
48         }
49     } else {
50         if (curr_p->next != NULL)
51             pthread_mutex_lock(&(curr_p->next->mutex));
52         else
53             rv = 0;
54         if (pred_p != NULL)
55             pthread_mutex_unlock(&(pred_p->mutex));
56         else
57             pthread_mutex_unlock(&head_mutex);
58         *pred_pp = curr_p;
59     }
```

```
58         *curr_pp = curr_p->next;
59         return rv;
60     }
61 }
62 void Init_ptrs(struct list_node_s curr_pp, struct list_node_s pred_pp) {
63     *pred_pp = NULL;
64     pthread_mutex_lock(&head_mutex);
65     *curr_pp = head;
66     if(*curr_pp != NULL)
67         pthread_mutex_lock(&((*curr_pp)->mutex));
68 }
69 int insert(int value) {
70     struct list_node_s* curr;
71     struct list_node_s* pred;
72     struct list_node_s* temp;
73     int rv = 1;
74
75     Init_ptrs(&curr, &pred);
76
77     while (curr != NULL && curr->data < value) {
78         Advance_ptrs(&curr, &pred);
79     }
80
81     if (curr == NULL || curr->data > value) {
82         temp = malloc(sizeof(struct list_node_s));
83         pthread_mutex_init(&(temp->mutex), NULL);
84         temp->data = value;
85         temp->next = curr;
86         if (curr != NULL)
87             pthread_mutex_unlock(&(curr->mutex));
88         if (pred == NULL) {
89             head = temp;
90             pthread_mutex_unlock(&head_mutex);
91         } else {
92             pred->next = temp;
93             pthread_mutex_unlock(&(pred->mutex));
94         }
95     } else {
96         if (curr != NULL)
97             pthread_mutex_unlock(&(curr->mutex));
98         if (pred != NULL)
99             pthread_mutex_unlock(&(pred->mutex));
100         else
101             pthread_mutex_unlock(&head_mutex);
102         rv = 0;
103     }
104
105     return rv;
```

---

106 }

---

#### ■ Análisis

- `struct list_node_s`:
  - Estructura que representa un nodo en la lista enlazada.
  - Contiene un campo para almacenar datos (`data`), un candado de mutex (`mutex`) para proteger el nodo y un puntero al siguiente nodo (`next`).

## Variables Globales

- `struct list_node_s* head`:
  - Puntero a la cabeza de la lista enlazada.
- `pthread_mutex_t head_mutex`:
  - Candado de mutex utilizado para proteger el acceso a la cabeza de la lista.

## Funciones

- `int member(int value)`:
  - Comprueba la membresía de un valor (`value`) en la lista enlazada.
  - Utiliza un candado de mutex para proteger el acceso a la cabeza de la lista y a cada nodo durante la búsqueda.
  - Retorna 1 si el valor está presente en la lista, 0 en caso contrario.
- `int insert(int value)`:
  - Inserta un nuevo valor (`value`) en la lista enlazada de manera ordenada.
  - Utiliza candados de mutex para garantizar la exclusión mutua durante la inserción.
  - Retorna 1 si la inserción es exitosa, 0 si el valor ya está presente en la lista.

## Funciones Auxiliares

- `int Advance_ptrs(struct list_node_s curr_pp, struct list_node_s pred_pp)`:
  - Avanza los punteros al nodo actual y al nodo anterior en la lista.
  - Utiliza candados de mutex para garantizar la exclusión mutua durante el avance.
  - Retorna 1 si el avance es exitoso, 0 si el nodo actual es el último, -1 si no hay nodos.
- `void Init_ptrs(struct list_node_s curr_pp, struct list_node_s pred_pp)`:
  - Inicializa los punteros al nodo actual y al nodo anterior para comenzar una operación en la lista.
  - Utiliza un candado de mutex para proteger el acceso a la cabeza de la lista durante la inicialización.

### 2.2.3. One Mutex per Node

- Implementación

---

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4  struct list_node_s {
5      int    data;
6      pthread_mutex_t mutex;
7      struct list_node_s* next;
8  };
9  struct list_node_s* head = NULL;
10 pthread_mutex_t head_mutex;
11
12 int member(int value) {
13     struct list_node_s *temp, *old_temp;
14     pthread_mutex_lock(&head_mutex);
15     temp = head;
16     if (temp != NULL) pthread_mutex_lock(&(temp->mutex));
17     pthread_mutex_unlock(&head_mutex);
18     while(temp != NULL && temp->data < value){
19         if (temp->next != NULL)
20             pthread_mutex_lock(&(temp->next->mutex));
21         old_temp = temp;
22         temp = temp->next;
23         pthread_mutex_unlock(&(old_temp->mutex));
24     }
25
26     if(temp == NULL || temp->data > value){
27         if (temp != NULL)
28             pthread_mutex_unlock(&(temp->mutex));
29         return 0;
30     }
31     else {
32         pthread_mutex_unlock(&(temp->mutex));
33         return 1;
34     }
35 }
36 int Advance_ptrs(struct list_node_s curr_pp, struct list_node_s pred_pp) {
37     int rv = 1;
38     struct list_node_s* curr_p = *curr_pp;
39     struct list_node_s* pred_p = *pred_pp;
40
41     if (curr_p == NULL) {
42         if (pred_p == NULL) {
43             pthread_mutex_unlock(&head_mutex);
44             return -1;
```

```
45         } else {
46             return 0;
47         }
48     } else {
49         if (curr_p->next != NULL)
50             pthread_mutex_lock(&(curr_p->next->mutex));
51         else
52             rv = 0;
53         if (pred_p != NULL)
54             pthread_mutex_unlock(&(pred_p->mutex));
55         else
56             pthread_mutex_unlock(&head_mutex);
57         *pred_pp = curr_p;
58         *curr_pp = curr_p->next;
59         return rv;
60     }
61 }
62 void Init_ptrs(struct list_node_s curr_pp, struct list_node_s pred_pp) {
63     *pred_pp = NULL;
64     pthread_mutex_lock(&head_mutex);
65     *curr_pp = head;
66     if(*curr_pp != NULL)
67         pthread_mutex_lock(&((*curr_pp)->mutex));
68 }
69 int insert(int value) {
70     struct list_node_s* curr;
71     struct list_node_s* pred;
72     struct list_node_s* temp;
73     int rv = 1;
74
75     Init_ptrs(&curr, &pred);
76
77     while (curr != NULL && curr->data < value) {
78         Advance_ptrs(&curr, &pred);
79     }
80
81     if (curr == NULL || curr->data > value) {
82         temp = malloc(sizeof(struct list_node_s));
83         pthread_mutex_init(&(temp->mutex), NULL);
84         temp->data = value;
85         temp->next = curr;
86         if (curr != NULL)
87             pthread_mutex_unlock(&(curr->mutex));
88         if (pred == NULL) {
89             head = temp;
90             pthread_mutex_unlock(&head_mutex);
91         } else {
92             pred->next = temp;
```

```
93         pthread_mutex_unlock(&(pred->mutex));
94     }
95 } else {
96     if (curr != NULL)
97         pthread_mutex_unlock(&(curr->mutex));
98     if (pred != NULL)
99         pthread_mutex_unlock(&(pred->mutex));
100     else
101         pthread_mutex_unlock(&head_mutex);
102     rv = 0;
103 }
104
105 return rv;
106 }
```

---

#### ■ Análisis

- `struct list_node_s`:
  - Estructura que representa un nodo en la lista enlazada.
  - Contiene un campo para almacenar datos (`data`), un candado de mutex (`mutex`) para proteger el nodo y un puntero al siguiente nodo (`next`).

## Variables Globales

- `struct list_node_s* head`:
  - Puntero a la cabeza de la lista enlazada.
- `pthread_mutex_t head_mutex`:
  - Candado de mutex utilizado para proteger el acceso a la cabeza de la lista.

## Funciones

- `int member(int value)`:
  - Comprueba la membresía de un valor (`value`) en la lista enlazada.
  - Utiliza candados de mutex para garantizar el acceso seguro a la cabeza de la lista y a cada nodo durante la búsqueda.
  - Retorna 1 si el valor está presente en la lista, 0 en caso contrario.
- `int insert(int value)`:
  - Inserta un nuevo valor (`value`) en la lista enlazada de manera ordenada.
  - Utiliza candados de mutex para garantizar la exclusión mutua durante la inserción.
  - Retorna 1 si la inserción es exitosa, 0 si el valor ya está presente en la lista.

## Funciones Auxiliares

- `int Advance_ptrs(struct list_node_s curr_pp, struct list_node_s pred_pp):`
  - Avanza los punteros al nodo actual y al nodo anterior en la lista.
  - Utiliza candados de mutex para garantizar la exclusión mutua durante el avance.
  - Retorna 1 si el avance es exitoso, 0 si el nodo actual es el último, -1 si no hay nodos.
- `void Init_ptrs(struct list_node_s curr_pp, struct list_node_s pred_pp):`
  - Inicializa los punteros al nodo actual y al nodo anterior para comenzar una operación en la lista.
  - Utiliza un candado de mutex para proteger el acceso a la cabeza de la lista durante la inicialización.

### 2.3. Diferencias entre las tres implementaciones

- Implementación Read-Write Locks :
  - Uso de Read-Write Locks para permitir operaciones de lectura concurrentes.
  - Operaciones de lectura y escritura están protegidas por locks para garantizar consistencia.
- Implementación One Mutex for Entire List:
  - Utiliza un único mutex global (`head_mutex`) para proteger el acceso a toda la lista enlazada.
  - Operaciones de lectura y escritura se bloquean mutuamente, limitando la concurrencia.
- Implementación One Mutex per Node:
  - Cada nodo de la lista tiene su propio mutex (`mutex`) para proteger operaciones en ese nodo específico.
  - Permite mayor concurrencia ya que operaciones en diferentes nodos pueden ocurrir simultáneamente.