

Parallel computing

Pthread

Objetivos

- Crear threads
- Sincronizar threads
- Acceder a secciones críticas
- Cache
- Usar Posix threads

Pthread

- Posix Thread Libreria multithreading
 - Sistemas Unix
- `STD::Thread` c++11
- Windows thread

Compiler

- `$ gcc -g -Wall -o hello hello.c -lpthread`

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4
5  /* Global variable: accessible to all threads */
6  int thread_count;
7
8  void* Hello(void* rank); /* Thread function */
9
10 int main(int argc, char* argv[]) {
11     long thread; /* Use long in case of a 64-bit system */
12     pthread_t* thread_handles;
13
14     /* Get number of threads from command line */
15     thread_count = strtol(argv[1], NULL, 10);
16
17     thread_handles = malloc (thread_count*sizeof(pthread_t));
18 }
```

```
19     for (thread = 0; thread < thread_count; thread++)
20         pthread_create(&thread_handles[thread], NULL,
21                       Hello, (void*) thread);
22
23     printf("Hello from the main thread\n");
24
25     for (thread = 0; thread < thread_count; thread++)
26         pthread_join(thread_handles[thread], NULL);
27
28     free(thread_handles);
29     return 0;
30 } /* main */
31
32 void* Hello(void* rank) {
33     long my_rank = (long) rank
34         /* Use long in case of 64-bit system */
35
36     printf("Hello from thread %ld of %d\n", my_rank,
37           thread_count);
38
39     return NULL;
40 } /* Hello */
```

Ejecutar

- `$./hello 4`
- Hello from the main thread
- Hello from thread 0 of 4
- Hello from thread 1 of 4
- Hello from thread 2 of 4
- Hello from thread 3 of 4

Codigo

- Variables globales (warning)
 - Ej. thread_count es compartida entre todos los threads
- Cada thread tiene su propio stack

Crear threads

- Objeto pthread_t por cada thread
 - Contiene informacion especifica de cada thread
 - Opaco (especifico al sistema)

```
int pthread_create(  
    pthread_t*      thread_p      /* out */,  
    const pthread_attr_t* attr_p   /* in */,  
    void*           (*start_routine)(void*) /* in */,  
    void*           arg_p         /* in */);
```

- pthread_t tiene que ser asignado con malloc antes de llamar esta funcion
- El tercer parámetro es la función que tendrá que ejecutar el thread
- El cuarto parámetro corresponde al parámetro que queremos mandar a la rutina

Crear threads

- Las funciones start_routine tienen que tener el siguiente prototipo
- `void* thread_function(void* args_p);`
- `void*` **cast cualquier tipo de puntero**
 - **struct para pasar varios datos**
- Podemos siempre pasar el rank de cada thread (util para debug)

```
int pthread_create(  
    pthread_t*      thread_p      /* out */,  
    const pthread_attr_t* attr_p    /* in */,  
    void*           (*start_routine)(void*) /* in */,  
    void*           arg_p          /* in */);
```


Crear threads

- El ejemplo anterior, los threads ejecutan todos el mismo código.
- **Cada thread podría ejecutar una función diferente.**

Ejecución de los threads

- El thread que ejecuta el main
 - **main thread**
- Mientras los threads creados con `pthread_create` ejecutan la función `hello`
 - Los threads retornan `NULL`
- No se especifica dónde y cuando se ejecuta el thread
- sobre un sistema cargado todos los threads podrían ejecutarse sobre el mismo núcleo
- En cambio sí un núcleo está libre el sistema lo aprovechara

Parar los threads

- Tenemos que llamar un `pthread_join` por cada thread creado

```
int pthread_join(  
    pthread_t  thread    /* in */,  
    void**     ret_val_p /* out */);
```

Ej. Multiplicación Matriz - Vector

a_{00}	a_{01}	\cdots	$a_{0,n-1}$
a_{10}	a_{11}	\cdots	$a_{1,n-1}$
\vdots	\vdots		\vdots
a_{i0}	a_{i1}	\cdots	$a_{i,n-1}$
\vdots	\vdots		\vdots
$a_{m-1,0}$	$a_{m-1,1}$	\cdots	$a_{m-1,n-1}$

x_0
x_1
\vdots
x_{n-1}

y_0
y_1
\vdots
$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1}$
\vdots
y_{m-1}

```
/* For each row of A */
for (i = 0; i < m; i++) {
    y[i] = 0.0;
    /* For each element of the row and each element of x */
    for (j = 0; j < n; j++)
        y[i] += A[i][j]* x[j];
}
```

Ej. Multiplicación Matriz - Vecctor

- Repartir el trabajo:

- ej: 3 threads, $n = 6$

Thread	Components of y
0	$y[0], y[1]$
1	$y[2], y[3]$
2	$y[4], y[5]$

- Cada thread ejecutara para todos sua “i” atribuidos :

```
y[i] = 0.0;  
for (j = 0; j < n; j++)  
    y[i] += A[i][j]*x[j];
```

Ej. Multiplicación Matriz - Vecctor

- y, x, A son compartidos
- calcular los “i” de cada thread “q”
- inicio = q * m/t
 - m = tamaño del vector resultante
 - t cuantos vectores
- fin = ((q+1)*m/t)-1

```
y[i] = 0.0;
for (j = 0; j < n; j++)
    y[i] += A[i][j]*x[j];
```

```
void* Pth_mat_vect(void* rank) {
    long my_rank = (long) rank;
    int i, j;
    int local_m = m/thread_count;
    int my_first_row = my_rank*local_m;
    int my_last_row = (my_rank+1)*local_m - 1;

    for (i = my_first_row; i <= my_last_row; i++) {
        y[i] = 0.0;
        for (j = 0; j < n; j++)
            y[i] += A[i][j]*x[j];
    }

    return NULL;
} /* Pth_mat_vect */
```

Secciones críticas

- En el ejemplo anterior no existe ninguna
 - race condition
- Como en OpenMP podemos estimar Pi

```
double factor = 1.0;
double sum = 0.0;
for (i = 0; i < n; i++, factor = -factor) {
    sum += factor/(2*i+1);
}
pi = 4.0*sum;
```

Secciones críticas

- Como en OpenMP podemos estimar Pi

```
1 void* Thread_sum(void* rank) {
2     long my_rank = (long) rank;
3     double factor;
4     long long i;
5     long long my_n = n/thread_count;
6     long long my_first_i = my_n*my_rank;
7     long long my_last_i = my_first_i + my_n;
8
9     if (my_first_i % 2 == 0) /* my_first_i is even */
10         factor = 1.0;
11     else /* my_first_i is odd */
12         factor = -1.0;
13
14     for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
15         sum += factor/(2*i+1);
16     }
17
18     return NULL;
19 } /* Thread_sum */
```

	<i>n</i>			
	10 ⁵	10 ⁶	10 ⁷	10 ⁸
π	3.14159	3.141593	3.1415927	3.14159265
1 Thread	3.14158	3.141592	3.1415926	3.14159264
2 Threads	3.14158	3.141480	3.1413692	3.14164686

Busy waiting

- sección crítica

- $x = x + y$

- 1 $y = \text{Compute}(\text{my rank});$

- 2 $\text{while } (\text{flag} \neq \text{my rank});$

- 3 $x = x + y;$

- 4 $\text{flag}++;$

Busy waiting

- Esta solución corrige los problemas
 - con $n=10^8$ el tiempo $p = 7 \cdot t_{\text{serial}}$

```
1 void* Thread_sum(void* rank) {
2     long my_rank = (long) rank;
3     double factor;
4     long long i;
5     long long my_n = n/thread_count;
6     long long my_first_i = my_n*my_rank;
7     long long my_last_i = my_first_i + my_n;
8
9     if (my_first_i % 2 == 0)
10         factor = 1.0;
11     else
12         factor = -1.0;
13
14     for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
15         while (flag != my_rank);
16         sum += factor/(2*i+1);
17         flag = (flag+1) % thread_count;
18     }
19
20     return NULL;
21 } /* Thread_sum */
```

Mutex

- Mutual exclusion
- Excluye los otros threads de una zona
- `pthread_mutex_t`
- `int pthread_mutex_init(pthread_mutex_t* mutex_p, NULL);`
- `int pthread_mutex_destroy(pthread_mutex_t* mutex_p);`

Mutex

- Acceder a una zona crítica (lock)
 - `int pthread_mutex_lock(pthread_mutex_t* mutex_p);`
- Dejar una zona crítica (unlock)
 - `int pthread_mutex_unlock(pthread_mutex_t* mutex_p);`
- lock
 - causa la **espera** de los demas threads
 - El sistema decide quién será el siguiente thread en obtener el acceso
 - No hay un orden predefinido
- unlock
 - notifica el sistema que la sección crítica está libre

Mutex

```
1 void* Thread_sum(void* rank) {
2     long my_rank = (long) rank;
3     double factor;
4     long long i;
5     long long my_n = n/thread_count;
6     long long my_first_i = my_n*my_rank;
7     long long my_last_i = my_first_i + my_n;
8     double my_sum = 0.0;
9
10    if (my_first_i % 2 == 0)
11        factor = 1.0;
12    else
13        factor = -1.0;
14
15    for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
16        my_sum += factor/(2*i+1);
17    }
18    pthread_mutex_lock(&mutex);
19    sum += my_sum;
20    pthread_mutex_unlock(&mutex);
21
22    return NULL;
23 } /* Thread_sum */
```

Mutex

- Cuando la cantidad de thread ≤ 8 (cantidad de cores)

$$\frac{T_{\text{serial}}}{T_{\text{parallel}}} \approx \text{thread_count}$$

- Eficiencia quasi idéal

- Cuando la cantidad de threads $>$ cores

- Busy waiting empeora, ej: 2 cores 5 threads (flag !!)

Time	flag	Thread				
		0	1	2	3	4
0	0	crit sect	busy-wait	susp	susp	susp
1	1	terminate	crit sect	susp	busy-wait	susp
2	2	—	terminate	susp	busy-wait	busy-wait
⋮	⋮			⋮	⋮	⋮
?	2	—	—	crit sect	susp	busy-wait

Threads	Busy-Wait	Mutex
1	2.90	2.90
2	1.45	1.45
4	0.73	0.73
8	0.38	0.38
16	0.50	0.38
32	0.80	0.40
64	3.56	0.38

SEMAPHORE

- Cómo obligar los thread a ejecutarse en una secuencia ?
 - Ej.: Operaciones no conmutativas como la multiplicación matricial

```
/* n and product_matrix are shared and initialized by the main  
thread */  
/* product_matrix is initialized to be the identity matrix */  
void* Thread_work(void* rank) {  
    long my_rank = (long) rank;  
    matrix_t my_mat = Allocate_matrix(n);  
    Generate_matrix(my_mat);  
    pthread_mutex_lock(&mutex);  
    Multiply_matrix(product_mat, my_mat);  
    pthread_mutex_unlock(&mutex);  
    Free_matrix(&my_mat);  
    return NULL;  
} /* Thread_work */
```

SEMAPHORE

- Cómo obligar los thread a ejecutarse en una secuencia ?
 - Ej.: Cada thread tiene que mandar un mensaje a otro thread
 - Thread 0 -> thread 1 -> thread 2 -> thread t -> thread 0
 - Un thread después de recibir su mensaje puede terminar
 - Array “messages” char** de tamaño t
 - Un thread agrega un mensaje messages[dest]=myMessage
 - Después lee el mensaje que recibió


SEMAPHORE

- Cómo obligar los thread a ejecutarse en una secuencia ?
 - Ej.: Cada thread tiene que mandar un mensaje a otro thread

```
1  /* messages has type char**. It's allocated in main. */
2  /* Each entry is set to NULL in main. */
3  void* Send_msg(void* rank) {
4      long my_rank = (long) rank;
5      long dest = (my_rank + 1) % thread_count;
6      long source = (my_rank + thread_count - 1) % thread_count;
7      char* my_msg = malloc(MSG_MAX*sizeof(char));
8
9      sprintf(my_msg, "Hello to %ld from %ld", dest, my_rank);
10     messages[dest] = my_msg;
11
12     if (messages[my_rank] != NULL)
13         printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
14     else
15         printf("Thread %ld > No message from %ld\n", my_rank,
16                source);
17     return NULL;
18 } /* Send_msg */
```

SEMAPHORE

- Cómo obligar los thread a ejecutarse en una secuencia ?
 - Ej.: Cada thread tiene que mandar un mensaje a otro thread
 - Es muy probable que el Thread 0 intenté leer su mensaje antes que el último thread le haya escrito



```
while (messages[my_rank] == NULL);  
printf("Thread %ld > %s\n", my_rank, messages[my_rank]);  
  
. . .  
messages[dest] = my_msg;  
Notify thread dest that it can proceed;  
  
Await notification from thread source  
printf("Thread %ld > %s\n", my_rank, messages[my_rank]);  
. . .
```

SEMAPHORE

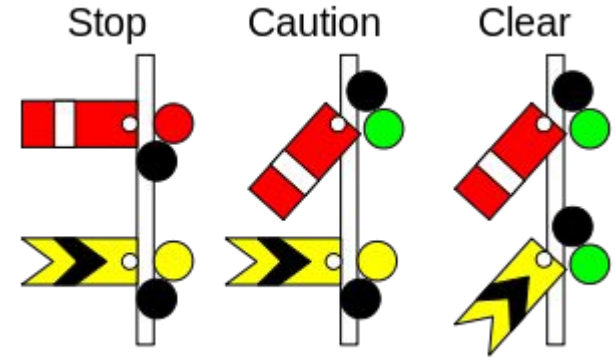
- Ej.: Cada thread tiene que mandar un mensaje a otro thread
- Podemos hacer esto con mutex ?

```
1  . . .
2  pthread_mutex_lock(mutex[dest]);
3  . . .
4  messages[dest] = my_msg;
5  pthread_mutex_unlock(mutex[dest]);
6  . . .
7  pthread_mutex_lock(mutex[my_rank]);
8  printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
9  . . .
```

- Thread 0 podría llegar a la linea 7 antes que thread t llegue a la linea 2

SEMAPHORE

- Solución: semaphores (Dijkstra)
 - Carriles de tren
 - Unsigned int
 - Muchas veces solo 0-1 (binary semaphore)
 - 0 locked - 1 unlocked
 - `sem_wait` bloquea si semaphore = 0 sino decrementa el semaphore
 - `sem_post` para incrementar el semaphore
 - los semphores no tienen “dueños” entonces el main thread puede inicializar todos los semaphores a 0



SEMAPHORE

- Solución: semaphores

```
1  /* messages is allocated and initialized to NULL in main */
2  /* semaphores is allocated and initialized to 0 (locked) in  
   main */
3  void* Send_msg(void* rank) {
4      long my_rank = (long) rank;
5      long dest = (my_rank + 1) % thread_count;
6      char* my_msg = malloc(MSG_MAX*sizeof(char));
7
8      sprintf(my_msg, "Hello to %ld from %ld", dest, my_rank);
9      messages[dest] = my_msg;
10     sem_post(&semaphores[dest])
        /* ''Unlock'' the semaphore of dest */
11
12     /* Wait for our semaphore to be unlocked */
13     sem_wait(&semaphores[my_rank]);
14     printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
15
16     return NULL;
17 } /* Send_msg */
```

SEMAPHORE

- Solución: semaphores
- `#include <semaphore.h>`

```
int sem_init(  
    sem_t*    semaphore_p    /* out */,  
    int        shared         /* in  */,  
    unsigned   initial_val    /* in  */);  
  
int sem_destroy(sem_t*    semaphore_p /* in/out */);  
int sem_post(sem_t*      semaphore_p /* in/out */);  
int sem_wait(sem_t*      semaphore_p /* in/out */);
```

Barreras

- Sincronizar los threads
 - Asegurar que todos lleguen al mismo lugar en el programa
 - **barrera**
 - **Ej. medir el tiempo del thread mas lento**

```
/* Shared */  
double elapsed_time;  
  
...  
/* Private */  
double my_start, my_finish, my_elapsed;  
  
...  
Synchronize threads;  
Store current time in my_start;  
/* Execute timed code */  
  
...  
Store current time in my_finish;  
my_elapsed = my_finish - my_start;  
  
elapsed = Maximum of my_elapsed values;
```

Barreras

- Sincronizar los threads
 - Asegurar que todos lleguen al mismo lugar en el programa
 - **barrera**
 - **Ej. depuración**

```
point in program we want to reach;  
barrier;  
if (my.rank == 0) {  
    printf("All threads reached this point\n");  
    fflush(stdout);  
}
```


Barreras

- Según los sistemas no disponen de alguna función de barrera
 - Hay que implementarla
 - busy-waiting o mutex
 - semaphore
 - variable de condition

Barreras

- Busy waiting y mutex
 - Contador de threads que pasan por una zona critica

```
/* Shared and initialized by the main thread */
int counter; /* Initialize to 0 */
int thread_count;
pthread_mutex_t barrier_mutex;
. . .

void* Thread_work(. . .) {
    . . .
    /* Barrier */
    pthread_mutex_lock(&barrier_mutex);
    counter++;
    pthread_mutex_unlock(&barrier_mutex);
    while (counter < thread_count);
    . . .
}
```

Barreras

- Se gasta mucho CPU en el while aun + si threads > cores
- Que pasa si más adelante se necesita otra barrera
 - no es posible resetear counter (o al menos muy peligroso)
 - Los threads se podrían quedar en el while

```
/* Shared and initialized by the main thread */
int counter; /* Initialize to 0 */
int thread_count;
pthread_mutex_t barrier_mutex;
. . .

void* Thread_work(. . .) {
    . . .
    /* Barrier */
    pthread_mutex_lock(&barrier_mutex);
    counter++;
    pthread_mutex_unlock(&barrier_mutex);
    while (counter < thread_count);
    . . .
}
```

Barreras

- Semaphore

```
sem_t count_sem;    /* Initialize to 1 */
sem_t barrier_sem;  /* Initialize to 0 */
. . .
void* Thread_work(...) {
    . . .
    /* Barrier */
    sem_wait(&count_sem);
    if (counter == thread_count-1) {
        counter = 0;
        sem_post(&count_sem);
        for (j = 0; j < thread_count-1; j++)
            sem_post(&barrier_sem);
    } else {
        counter++;
        sem_post(&count_sem);
        sem_wait(&barrier_sem);
    }
    . . .
}
```

Barreras

- 2 Semaphores
 - count_sem: protege counter 1
 - barrier_sem: impide los threads de terminar 0
- Cuando un thread entra en el “else”
 - incrementa el contador libera count_sem
 - se queda a la espera de su liberación barrier_sem
- Cuando el ultimo thread entra
 - resetea el contador
 - libera el count_sem
 - libera todo los threads uno por uno

```
sem_t count_sem;    /* Initialize to 1 */
sem_t barrier_sem;  /* Initialize to 0 */
. . .
void* Thread_work(...) {
    . . .
    /* Barrier */
    sem_wait(&count_sem);
    if (counter == thread_count-1) {
        counter = 0;
        sem_post(&count_sem);
        for (j = 0; j < thread_count-1; j++)
            sem_post(&barrier_sem);
    } else {
        counter++;
        sem_post(&count_sem);
        sem_wait(&barrier_sem);
    }
    . . .
}
```

Barreras

- el for libera todos los threads llamando a `sem_post`
 - `sem_post` incrementa el `count`
 - no importa si se llama muchas veces `sem_post` sin `sem_wait`
 - `sem_wait` decreuenta hasta 0
- No ocupa el CPU por gusto
- Podemos reutilizar la barrera ?
- reinicializamos el contador OK
- `count_sem` a 1 OK
- `barrier_sem` **No estamos seguros**

```
sem_t count_sem;    /* Initialize to 1 */
sem_t barrier_sem;  /* Initialize to 0 */

...
void* Thread_work(...) {
    ...
    /* Barrier */
    sem_wait(&count_sem);
    if (counter == thread_count-1) {
        counter = 0;
        sem_post(&count_sem);
        for (j = 0; j < thread_count-1; j++)
            sem_post(&barrier_sem);
    } else {
        counter++;
        sem_post(&count_sem);
        sem_wait(&barrier_sem);
    }
    ...
}
```

Barreras

- Variable condición **pthread_cond_t**
- Permite la suspensión de un thread hasta que un evento o condición ocurra
- Asocia a un mutex
- Desbloquear un thread bloqueado
 - `int pthread_cond_signal(pthread_cond_t* cond_var_p);`
- Desbloquea todo los threads
 - `int pthread_cond_broadcast(pthread_cond_t* cond_var_p);`
- Desbloquea un mutex
 - `int pthread_cond_wait(pthread_cond_t* cond_var_p, pthread_mutex_t* mutex_p);`
 - Esto causa que el thread ejecutando se bloquee hasta ser desbloqueado por otro thread (broadcast o signal)

```
pthread_mutex_unlock(&mutex_p);  
wait-on-signal(&cond-var-p);  
pthread_mutex_lock(&mutex_p);
```

Barreras

- Un thread entra en el else y llama a `cond_wait`
 - el thread se queda esperando y se desbloquea el mutex
 - así para todos menos el último
- El último thread entra en el if
 - reinicializa el contador
 - Desbloquea todos los threads uno por uno

```
/* Shared */
int counter = 0;
pthread_mutex_t mutex;
pthread_cond_t cond_var;

...
void* Thread-work(. . .) {
    ...
    /* Barrier */
    pthread_mutex_lock(&mutex);
    counter++;
    if (counter == thread-count) {
        counter = 0;
        pthread_cond_broadcast(&cond_var);
    } else {
        pthread_cond_wait(&cond_var, &mutex)
    }
    pthread_mutex_unlock(&mutex);
    ...
}
```


Read-write locks

- Compartir una estructura de datos entre threads
- Ej. lista enlazada
 - member, insert, delete

```
struct list_node_s {  
    int data;  
    struct list_node_s* next;  
}
```



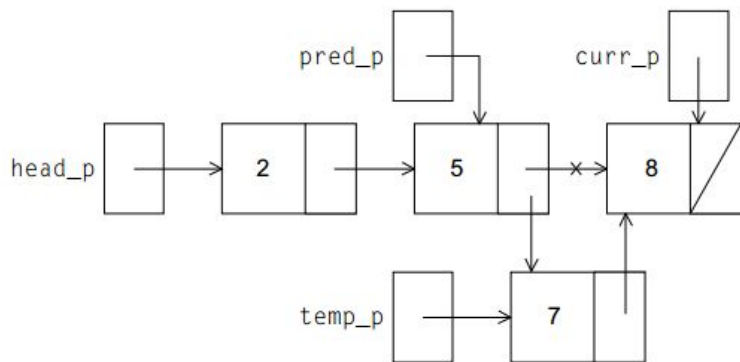
Read-write locks

- Member

```
1  int  Member(int value, struct list_node_s* head_p) {  
2      struct list_node_s* curr_p = head_p;  
3  
4      while (curr_p != NULL && curr_p->data < value)  
5          curr_p = curr_p->next;  
6  
7      if (curr_p == NULL || curr_p->data > value) {  
8          return 0;  
9      } else {  
10         return 1;  
11     }  
12 }  /* Member */
```

Read-write locks

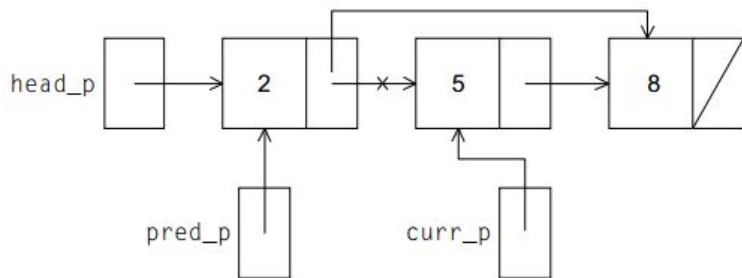
- Insert



```
1 int Insert(int value, struct list_node_s** head_p) {
2     struct list_node_s* curr_p = *head_p;
3     struct list_node_s* pred_p = NULL;
4     struct list_node_s* temp_p;
5
6     while (curr_p != NULL && curr_p->data < value) {
7         pred_p = curr_p;
8         curr_p = curr_p->next;
9     }
10
11     if (curr_p == NULL || curr_p->data > value) {
12         temp_p = malloc(sizeof(struct list_node_s));
13         temp_p->data = value;
14         temp_p->next = curr_p;
15         if (pred_p == NULL) /* New first node */
16             *head_p = temp_p;
17         else
18             pred_p->next = temp_p;
19         return 1;
20     } else { /* Value already in list */
21         return 0;
22     }
23 }
```

Read-write locks

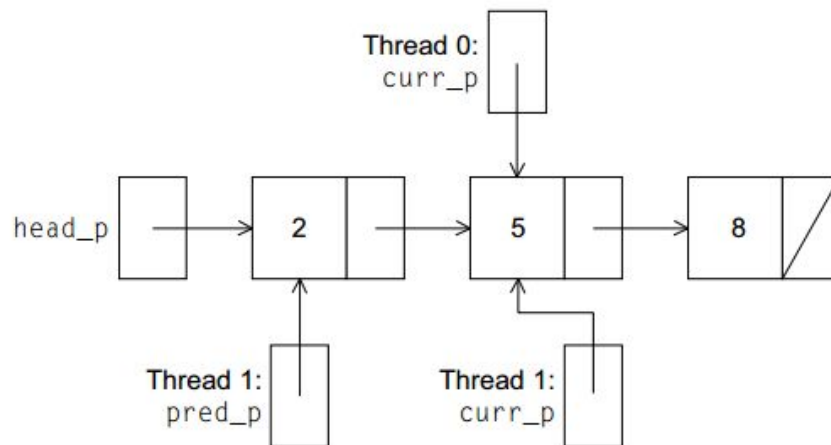
- delete



```
1 int Delete(int value, struct list_node_s** head_p) {
2     struct list_node_s* curr_p = *head_p;
3     struct list_node_s* pred_p = NULL;
4
5     while (curr_p != NULL && curr_p->data < value) {
6         pred_p = curr_p;
7         curr_p = curr_p->next;
8     }
9
10    if (curr_p != NULL && curr_p->data == value) {
11        if (pred_p == NULL) { /* Deleting first node in list */
12            *head_p = curr_p->next;
13            free(curr_p);
14        } else {
15            pred_p->next = curr_p->next;
16            free(curr_p);
17        }
18        return 1;
19    } else { /* Value isn't in list */
20        return 0;
21    }
22 } /* Delete */
```

Read-write locks

- Que pasa cuando usamos esta estructura con varios threads
- Ej. Thread 0 member(5), thread 1 delete(5)



Read-write locks

- Que pasa cuando usamos esta estructura con varios threads
- Ej. Thread 0 member(5), thread 1 delete(5)
 - thread 0 retorne True mientras no existe el 5
 - thread 0 quiera acceder al 5 mientras la memoria ha sido liberada (crash)
- Solución rapida

```
Pthread_mutex_lock(&list_mutex);  
Member(value);  
Pthread_mutex_unlock(&list_mutex);
```

Read-write locks

- Solución 1 : los threads acceden a la lista uno a la vez
- Solución 2 : los threads acceden a un nodo uno a la vez
- Alternativa : **read-write-lock**
 - **Dos funciones de lock**
 - **read**
 - **write**
 - **Múltiples** threads pueden obtener el lock para el **read**
 - **Uno** solamente puede obtener el lock para **write**
 - Si **un thread** ha obtenido el lock para el **read** entonces **ninguno** obtendrá un lock para el **writing**
 - Si **un thread** ha obtenido el lock para el **write** entonces **ninguno** obtendrá un lock para el **writing/reading**

Read-write locks

- **read-write-lock**

- `int pthread_rwlock_init(pthread_rwlock_t* rwlock_p, NULL);`
- `int pthread_rwlock_destroy(pthread_rwlock_t* rwlock_p);`
- `int pthread_rwlock_rdlock(pthread_rwlock_t* rwlock_p);`
- `int pthread_rwlock_wrlock(pthread_rwlock_t* rwlock_p);`
- `int pthread_rwlock_unlock(pthread_rwlock_t* rwlock_p);`

```
pthread_rwlock_rdlock(&rwlock);  
Member(value);  
pthread_rwlock_unlock(&rwlock);  
...  
pthread_rwlock_wrlock(&rwlock);  
Insert(value);  
pthread_rwlock_unlock(&rwlock);  
...  
pthread_rwlock_wrlock(&rwlock);  
Delete(value);  
pthread_rwlock_unlock(&rwlock);
```


Read-write locks

- Performance
 - list size 1000
 - 100 000 ops
 - 99.9% member 0.05% insert 0.05% delete

Implementation	Number of Threads			
	1	2	4	8
Read-Write Locks	0.213	0.123	0.098	0.115
One Mutex for Entire List	0.211	0.450	0.385	0.457
One Mutex per Node	1.680	5.700	3.450	2.700

Read-write locks

- Performance
 - list size 1000
 - 100 000 ops
 - 80% member 10% insert 10% delete

Implementation	Number of Threads			
	1	2	4	8
Read-Write Locks	2.48	4.97	4.69	4.71
One Mutex for Entire List	2.50	5.13	5.04	5.11
One Mutex per Node	12.00	29.60	17.00	12.00

thread c++11

- Crear thread
 - `std::thread t1(call_from_thread);`
- Join
 - `t1.join()`

```
void call_from_thread(int tid) {  
    std::cout << "Launched by thread " << tid << std::endl;  
}
```

```
int main() {  
  
    std::thread t[num_threads];  
    for (int i = 0; i < num_threads; ++i) {  
        t[i] = std::thread(call_from_thread, i);  
    }
```