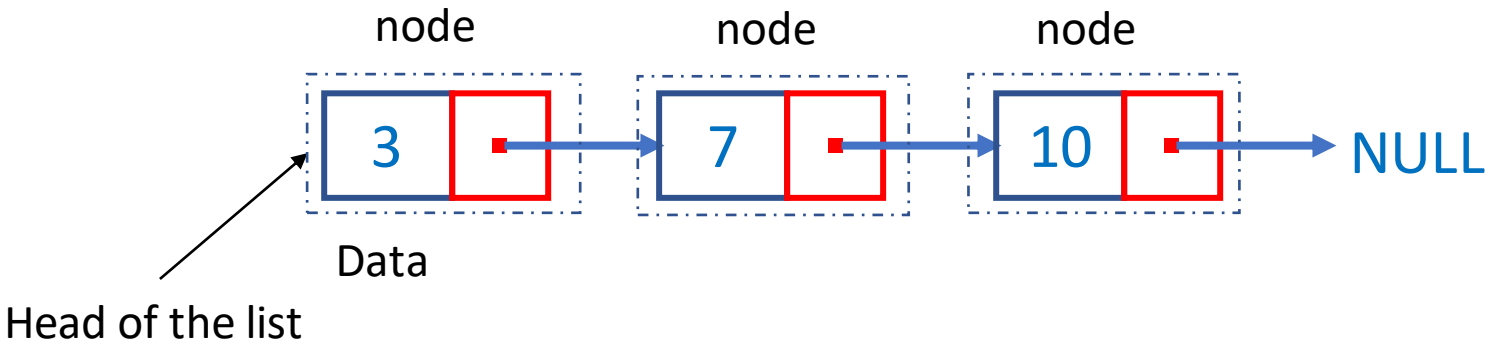
Application:
Concurrent operations on a shared Linked List

Application: Concurrent operations on a shared Linked List

Consider a sorted linked list of integers with the following operations:

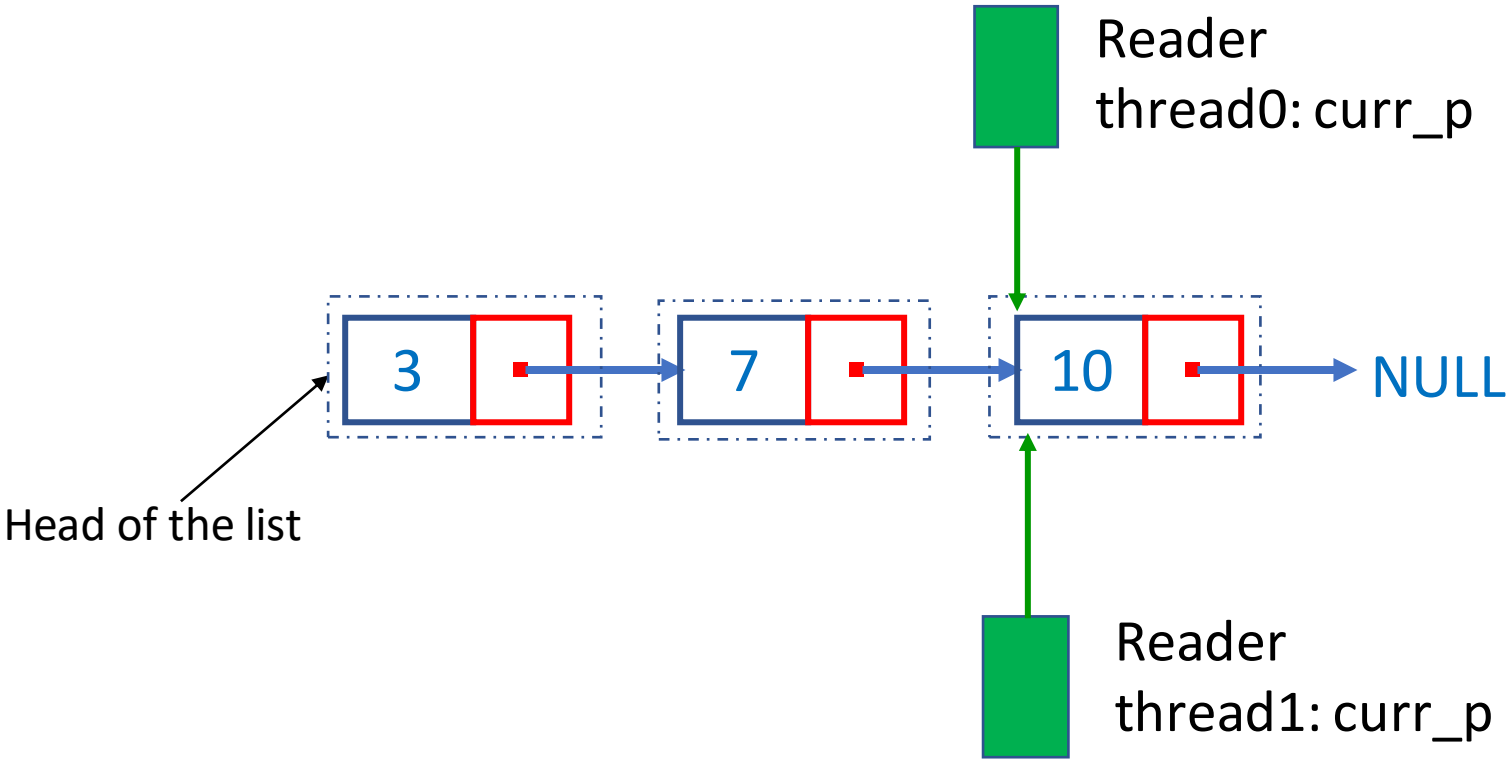
- Insert: inserts a new node maintaining the sorted order.
- Delete: deletes an existing node.
- Member: returns true/false depending on node present/absent.



Challenge: Multiple concurrent threads perform these operations on a shared linked list.

Simultaneous access by two threads

Two reader threads in operation.

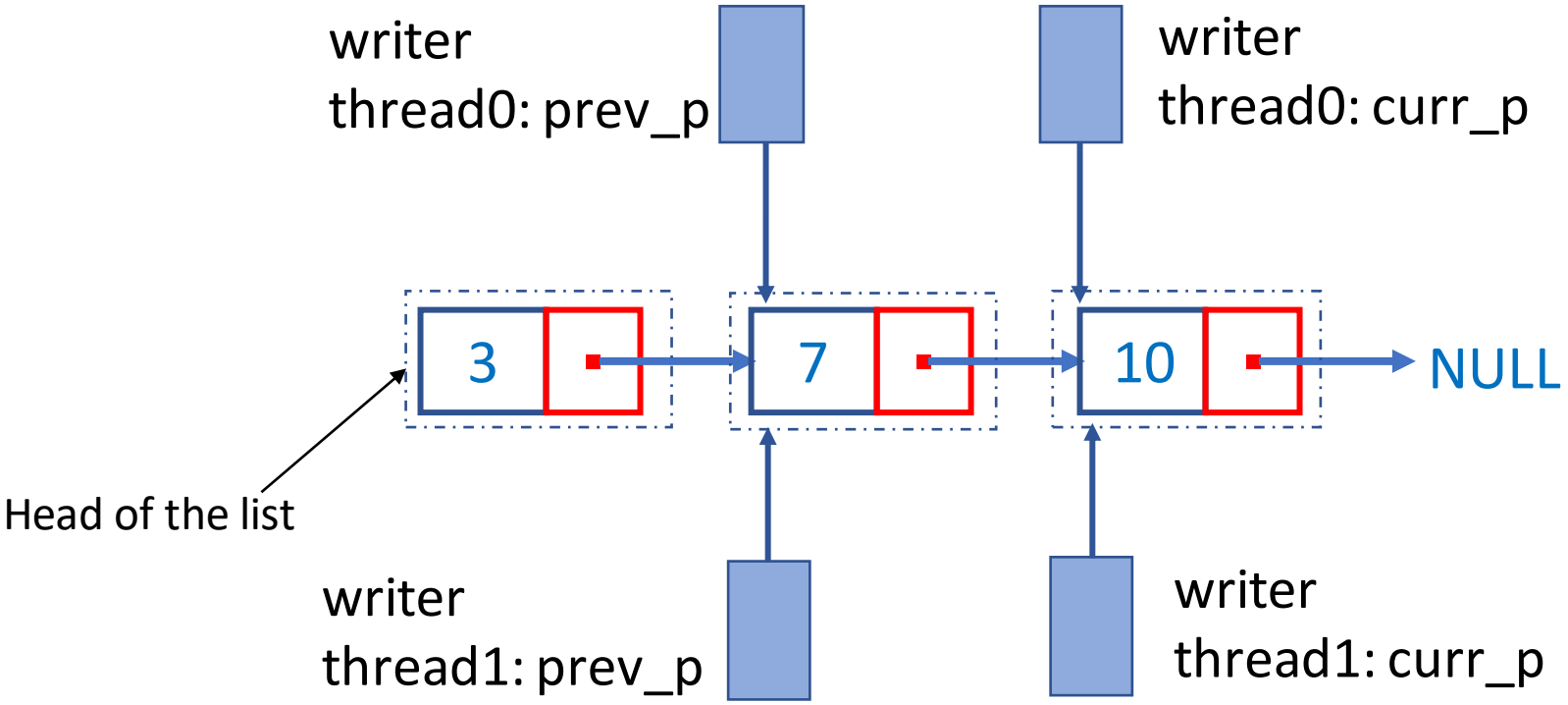


Multiple threads **can read** concurrently.

Simultaneous access by two threads

Two concurrent operations:

- Thread0 wants to insert node value 8 in the list.
- Thread1 wants to insert node value 9 in the list.

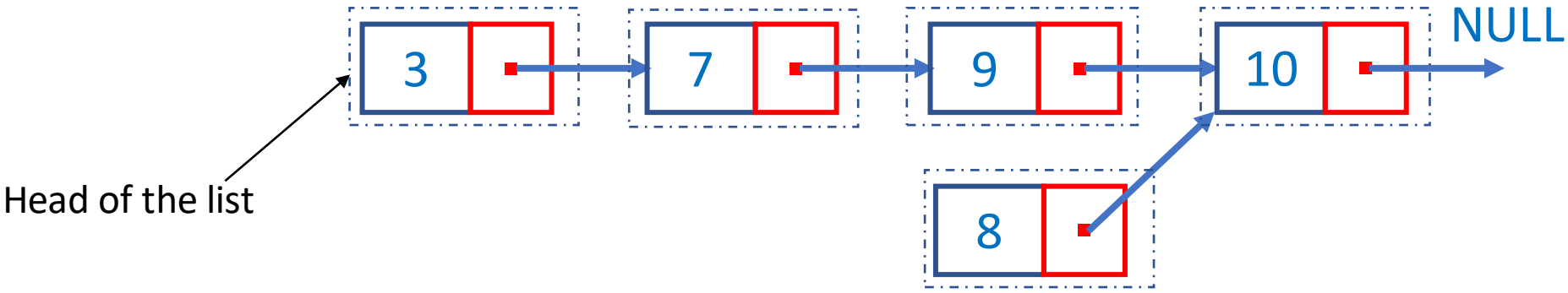


Multiple threads **cannot write** concurrently.

Simultaneous access by two threads

Two concurrent operations:

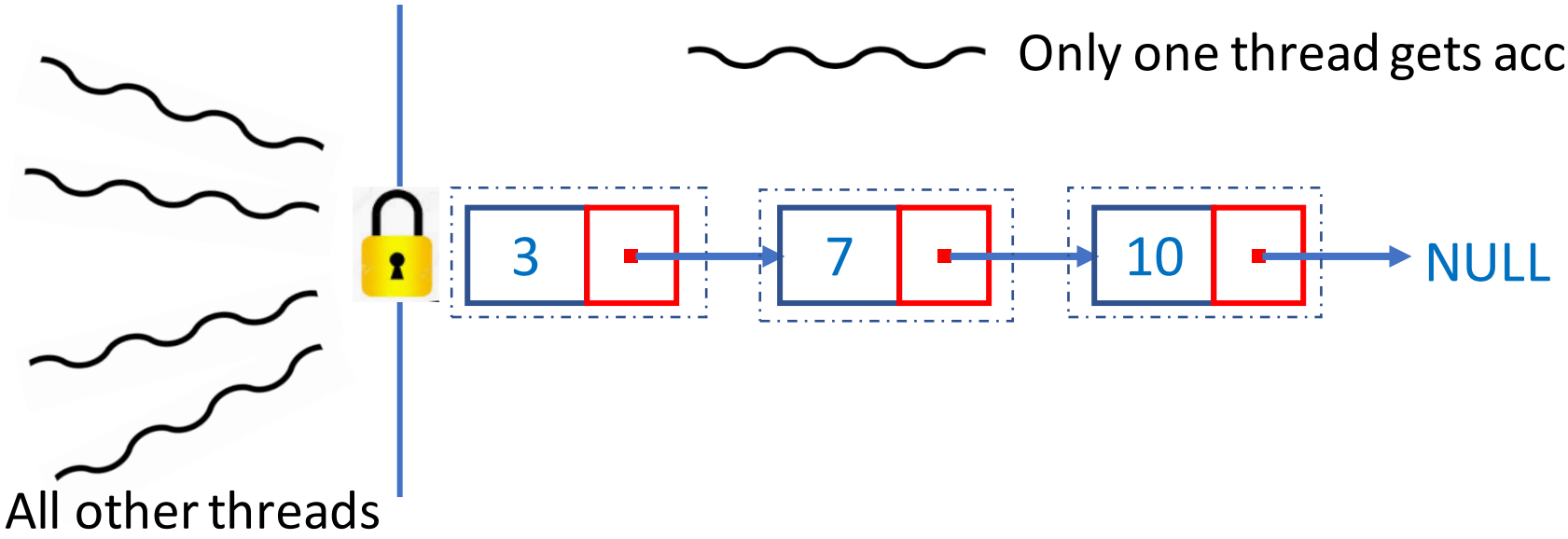
- Thread0 wants to insert node value 8 in the list.
- Thread1 wants to insert node value 9 in the list.



Multiple threads **cannot write** concurrently.

Solution1: Only one thread can access the list

- A naive solution is to simply lock the entire list to serialize access to the list.
- Serialization of access can be implemented using a mutex.




```
...
pthread_mutex_lock( &mutex1 );
 member(value0);
pthread_mutex_unlock( &mutex1 );
```

Thread0

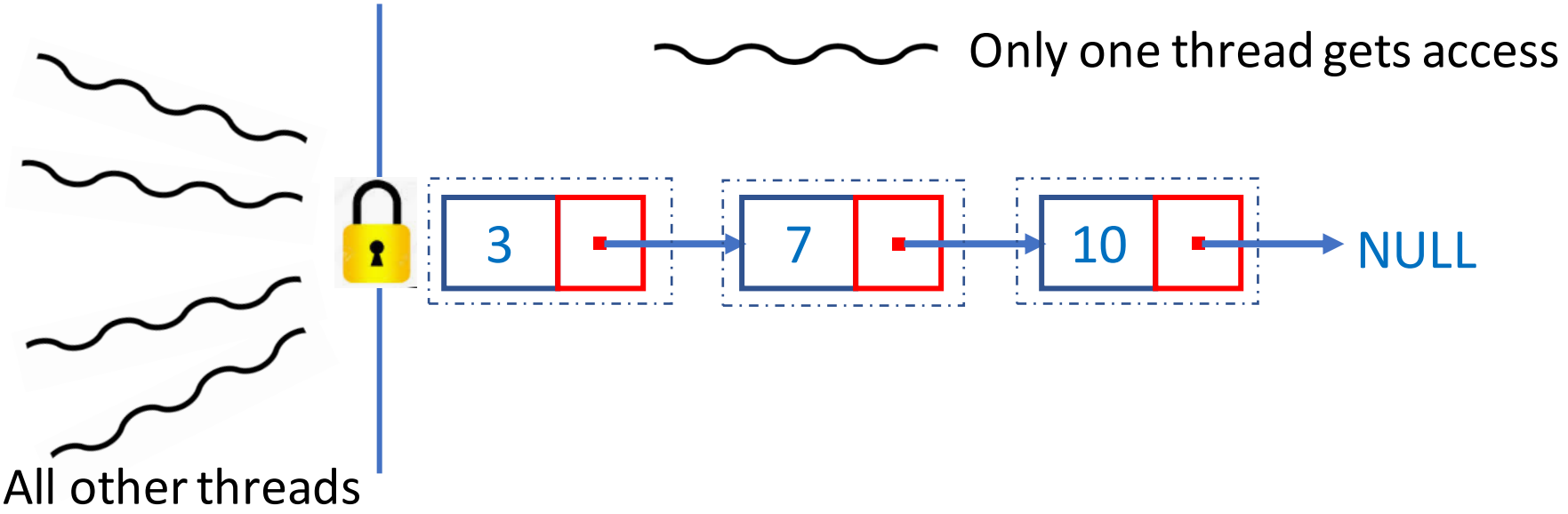
```
...
pthread_mutex_lock( &mutex1 );
 insert(value1);
pthread_mutex_unlock( &mutex1 );
```

Thread1

. . .

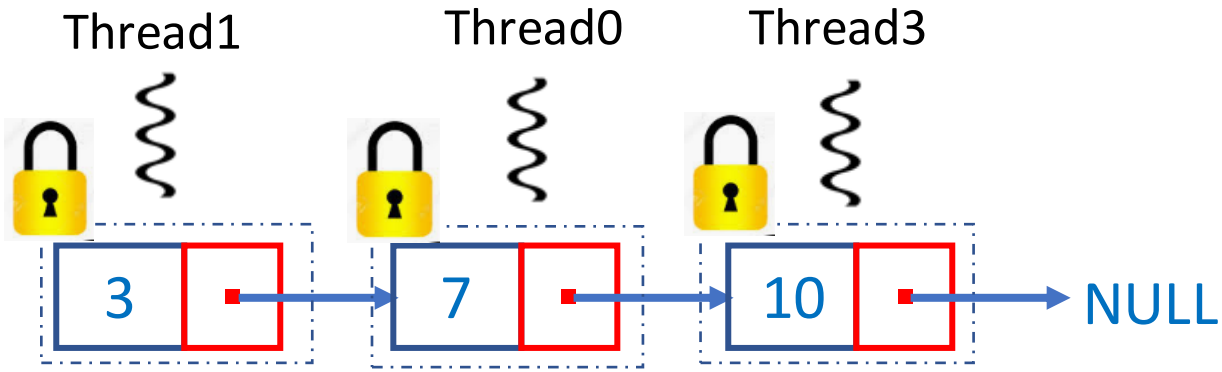
Issues with Solution1

- Only one thread gets access to the list.
- If vast majority of operations are ‘read’, then this approach fails to exploit parallelism.
- On the other hand, if most of the operations are ‘write’ then this approach may be the best and easy solution.



Solution2: Granular access to individual nodes

- Instead of locking the entire list, lock individual nodes.
- This gives granular access to the nodes.
  Example: Thread-M accesses one node while Thread-N accesses another node.



- Implementation requires one mutex lock per node.

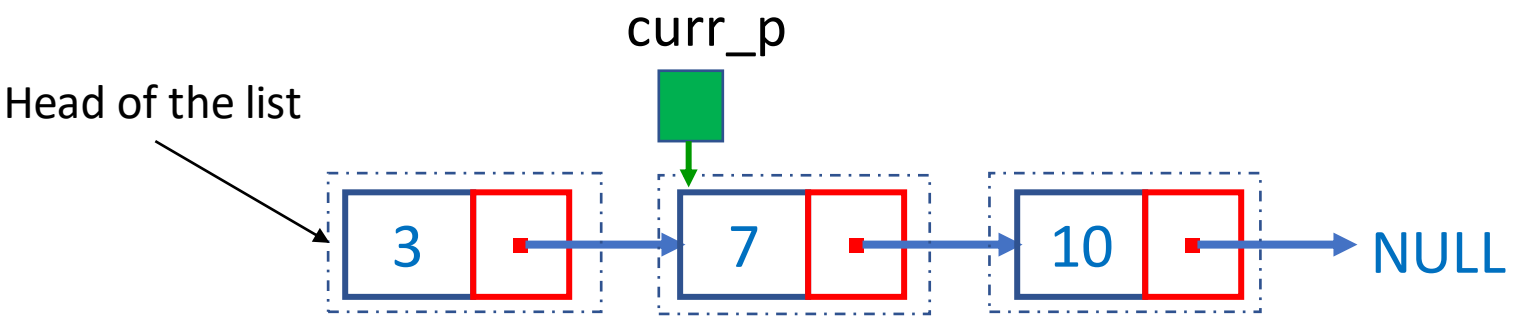
```
typedef struct Node{
        int data;
        struct Node *next;
        pthread_mutex_t mutex;
} Node;
```

With Solution2, code becomes a lot more complicated

Non-threaded Linked list: function Member()

Member() returns 1 if a node with the input 'value' is present in the list. Otherwise, it returns 0;

```
int Member(list *l, int value){
        Node *curr_p = l->head;
        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;
}
```



Implementation of Member() for Solution2

```
int Member(int value){
        Node *curr_p;

        pthread_mutex_lock(&head_mutex);
        curr_p = head;
        while(curr_p!=NULL && curr_p->data < value){
                if(curr_p->next != NULL)
                        pthread_mutex_lock(&(curr_p->next->mutex));
                if(curr_p == head)
                        pthread_mutex_unlock(&head_mutex);

                pthread_mutex_unlock(&(curr_p->mutex));
                curr_p = curr_p->next;
        }

        // Remaining part in the next slide
```

Implementation of Member() for Solution2

```
        // continuation from the previous slide
        if(curr_p == NULL || curr_p->data > value){
                if(curr_p == head)
                        pthread_mutex_unlock(&head_mutex);
                if(curr_p != NULL)
                        pthread_mutex_unlock(&(curr_p->mutex));
                return 0;
        }
        else{
                if(curr_p == head)
                        pthread_mutex_unlock(&head_mutex);
                pthread_mutex_unlock(&(curr_p->mutex));
                return 1;
        }
}
```

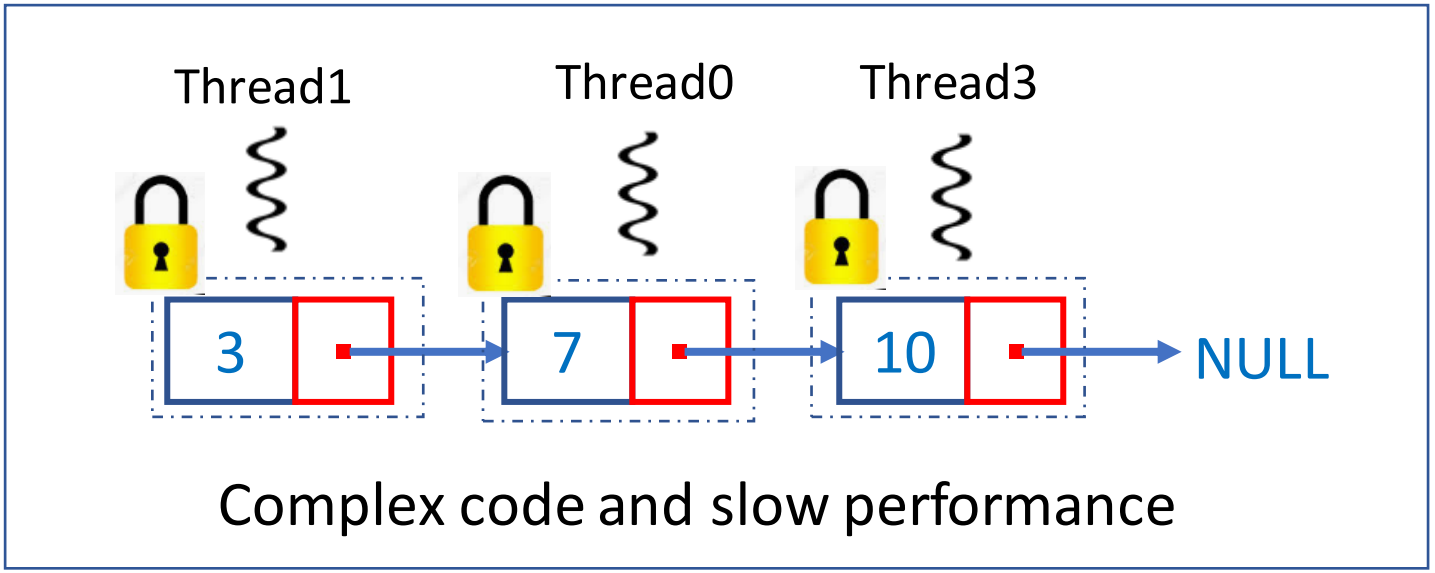
Source code of threaded linked list with one mutex per node is available at https://www.csee.umbc.edu/~tsimo1/CMSC483/cs220/code/pth-rw/pth_linked_list_mult_mut.c

Issues with Solution2: Granular access to individual nodes

- With Solution2, simple Member() function becomes rather 'complex'.

- Every time a thread tries to access a node, it needs to
  → check if a mutex lock is available
  → then locking and unlocking of the mutex lock etc.

- Performance will be much slower.



Solution1 and Solution2: summary

- The first solution only allows one thread to access the entire list at any instant.
  → defeats the purpose of multi-threading

- The second only allows one thread to access any given node at any instant.
  → major performance problem and complicated code.

Can we have a simpler and more efficient multi-threaded linked list?

Pthreads provide another kind of lock known as ‘read-write lock’.

Read-write locks

- A read-write lock is declared and initialized as

```
pthread_rwlock_t lock = PTHREAD_RWLOCK_INITIALIZER;
```

- A read-write lock is somewhat like a mutex except that it provides two lock functions.
  - for just reading

```
pthread_rwlock_rdlock(&lock);
```

  - for read-write access

```
pthread_rwlock_wrlock(&lock);
```

  - There is only one unlock function

```
pthread_rwlock_unlock(&lock);
```

Rules that read-write locks follow

Goal: allow multiple threads to read,
but allow only one thread to write.

```
pthread_rwlock_rdlock(){
        If no other thread holds the lock, then get the lock.
        Else if other threads hold the read-lock, then get the lock.
        Else if another thread holds the write-lock, then wait.
}
```

```
pthread_rwlock_wrlock(){
        If no other threads hold the read or write lock, then get the lock.
        Else, wait for the lock.
}
```

Application of read-write lock to Linked list

```
...
pthread_rwlock_rdlock(&lock);
Member(value1);
pthread_rwlock_unlock(&lock);

pthread_rwlock_wrlock(&lock);
Insert(value2);
pthread_rwlock_unlock(&lock);
...
```

One thread

```
...
pthread_rwlock_wrlock(&lock);
Delete(value3);
pthread_rwlock_unlock(&lock);

pthread_rwlock_rdlock(&lock);
Member(value3);
pthread_rwlock_unlock(&lock);
...
```

Another concurrent thread

...

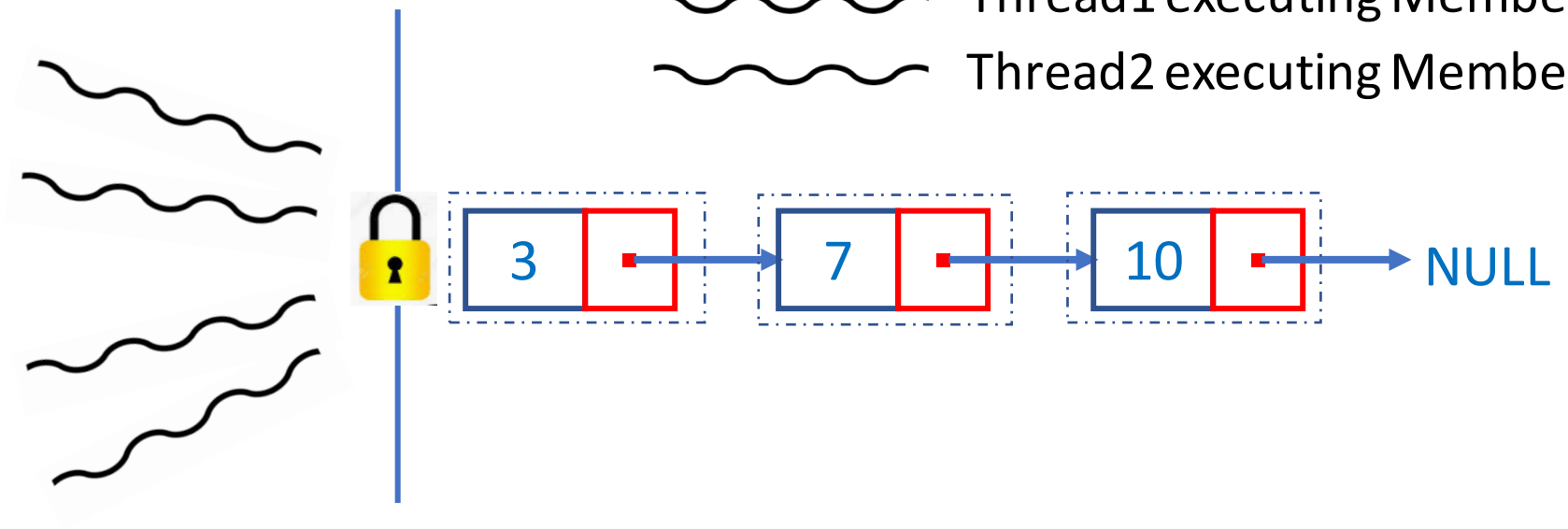
Multiple concurrent threads perform operations on the linked list.

- Read lock is used for functions that do not modify the list.
- Write lock is used for functions that modify the list.

Application of read-write lock to Linked list

Writer threads wait for reader threads to finish.

Multiple reader threads get concurrent access.



Application of read-write lock to Linked list

Other threads wait.

Only one writer thread gets exclusive access.