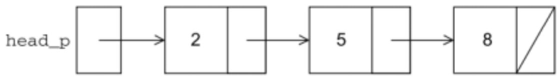
Read-write locks

Let's take a look at the problem of controlling access to a large, shared data structure, which can be either simply searched or updated by the threads. For the sake of explicitness, let's suppose the shared data structure is a sorted, singly-linked list of ints, and the operations of interest are Member, Insert, and Delete.

- The list itself is composed of a collection of list nodes, each of which is a struct with two members: an int and a pointer to the next node.

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

- Functions:

```
int  Member(int value, struct list_node_s* head_p) {
   struct list_node_s* 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;
   }
}  /* Member */
```

Program 4.9: The Member function.

```
int Insert(int value, struct list_node_s** head_pp) {
   struct list_node_s* curr_p = *head_pp;
   struct list_node_s* pred_p = NULL;
   struct list_node_s* temp_p;

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

   if (curr_p == NULL || curr_p->data > value) {
      temp_p = malloc(sizeof(struct list_node_s));
      temp_p->data = value;
      temp_p->next = curr_p;
      if (pred_p == NULL)  /* New first node */
         *head_pp = temp_p;
      else
         pred_p->next = temp_p;
      return 1;
   } else { /* Value already in list */
      return 0;
   }
}  /* Insert */
```

Program 4.10: The Insert function.

```
int Delete(int value, struct list_node_s** head_pp) {
   struct list_node_s* curr_p = *head_pp;
   struct list_node_s* pred_p = NULL;

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

   if (curr_p != NULL && curr_p->data == value) {
      if (pred_p == NULL) { /* Deleting first node in list */
         *head_pp = curr_p->next;
         free(curr_p);
      } else {
         pred_p->next = curr_p->next;
         free(curr_p);
      }
      return 1;
   } else { /* Value isn't in list */
      return 0;
   }
} /* Delete */
```

Program 4.11: The `Delete` function.

A multithreaded linked list

- Now let's try to use these functions in a Pthreads program. To share access to the list, we can define head_p to be a global variable. This will simplify the function headers for Member, Insert, and Delete, since we won't need to pass in either head_p or a pointer to head_p, we'll only need to pass in the value of interest
- Since multiple threads can simultaneously read a memory location without conflict, it should be clear that multiple threads can simultaneously execute Member.
- On the other hand, Delete and Insert also write to memory locations, so there may be problems if we try to execute either of these operations at the same time as another operation.
  - As an example, suppose that thread 0 is executing Member(5) at the same time that thread 1 is executing Delete(5), and the current state of the list is shown in Fig. 4.7. An obvious problem is that if thread 0 is executing Member(5), it is going to report that 5 is in the list, when, in fact, it may be deleted even before thread 0 returns.
  - A second obvious problem is if thread 0 is executing Member(8), thread 1 may free the memory used for the node storing 5 before thread 0 can advance to the node storing 8.
  - It's OK for multiple threads to simultaneously execute Member—that is, read the list nodes—but it's unsafe for multiple threads to access the list if at least one of the threads is executing an Insert or a Delete—that is, is writing to the list nodes
- How can we deal with this problem?
  - An obvious solution is to simply lock the list any time that a thread attempts to access it

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

    - An equally obvious problem with this solution is that we are serializing access to the list, and if the vast majority of our operations are calls to Member, we'll fail to exploit this opportunity for parallelism.
    - Inefficient and doesn't parallelize like we want it to
  - An alternative to this approach involves "finer-grained" locking. Instead of locking the entire list, we could try to lock individual nodes.

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

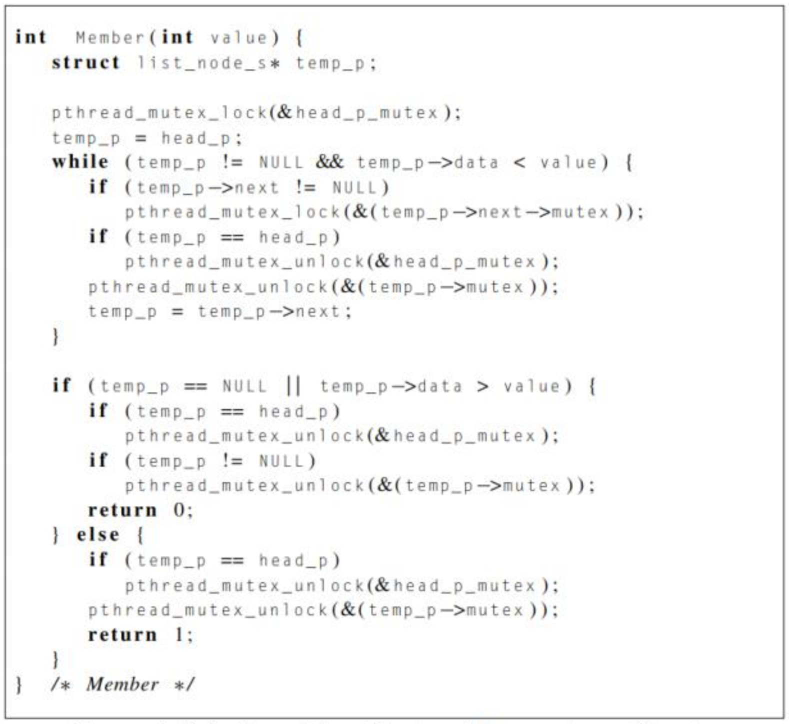
    Note that this will also require that we have a mutex associated with the head_p pointer.

    - This is a much better idea but creating a mutex for each node will severely increase the size of each node and consequently the size of our list
    - With this solution, our member() function would now look like this:

    ```
    int Member(int value) {
    ```

    Too bulky and complex, as well

- With this solution, our member() function would now look like this:



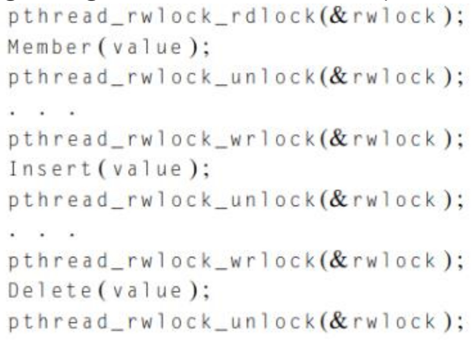
```
int  Member(int value) {
   struct list_node_s* temp_p;

   pthread_mutex_lock(&head_p_mutex);
   temp_p = head_p;
   while (temp_p != NULL && temp_p->data < value) {
      if (temp_p->next != NULL)
         pthread_mutex_lock(&(temp_p->next->mutex));
      if (temp_p == head_p)
         pthread_mutex_unlock(&head_p_mutex);
      pthread_mutex_unlock(&(temp_p->mutex));
      temp_p = temp_p->next;
   }

   if (temp_p == NULL || temp_p->data > value) {
      if (temp_p == head_p)
         pthread_mutex_unlock(&head_p_mutex);
      if (temp_p != NULL)
         pthread_mutex_unlock(&(temp_p->mutex));
      return 0;
   } else {
      if (temp_p == head_p)
         pthread_mutex_unlock(&head_p_mutex);
      pthread_mutex_unlock(&(temp_p->mutex));
      return 1;
   }
}  /* Member */
```

Program 4.12: Implementation of Member with one mutex per list node.

Too bulky and complex, as well as its not as efficient as we want it to be

Pthreads read-write locks

- Neither of our multithreaded linked lists exploits the potential for simultaneous access to any node by threads that are executing Member.
  - The first solution only allows one thread to access the entire list at any instant,
  - and the second only allows one thread to access any given node at any instant.
- An alternative is provided by Pthreads' read-write locks.
  - A read-write lock is somewhat like a mutex except that it provides two lock functions.
    - The first lock function locks the read-write lock for reading,
    - while the second locks it for writing.
  - Multiple threads can thereby simultaneously obtain the lock by calling the read-lock function, while only one thread can obtain the lock by calling the write-lock function.
    - Thus if any threads own the lock for reading, any threads that want to obtain the lock for writing will block in the call to the writelock function.
    - Furthermore, if any thread owns the lock for writing, any threads that want to obtain the lock for reading or writing will block in their respective locking functions.
  - Essentially we only have two phases: reading and writing and we can lock variables, functions, etc with these
  - Using Pthreads read-write locks, we can protect our linked list functions with the following code (we're ignoring function return values):



```
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);
```

- Pthread lock and unlock functions:

```
int pthread_rwlock_rdlock(
   pthread_rwlock_t*  rwlock_p  /* in/out */);

int pthread_rwlock_wrlock(
   pthread_rwlock_t*  rwlock_p  /* in/out */);

int pthread_rwlock_unlock(
   pthread_rwlock_t*  rwlock_p  /* in/out */);
```

- We should initialize locks before using them and destroy them after use

```
int pthread_rwlock_init(
        pthread_rwlock_t*              rwlock_p   /* out */,
        const pthread_rwlockattr_t*    attr_p     /* in  */);

/* And, the static version: */
pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;


int pthread_rwlock_destroy(
   pthread_rwlock_t*  rwlock_p  /* in/out*/);
```

What is the best solution of out the three?

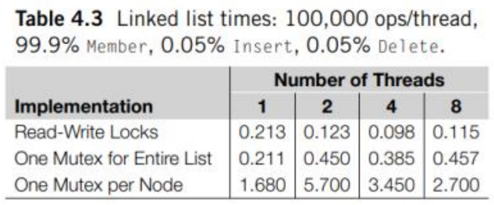


Table 4.3 Linked list times: 100,000 ops/thread, 99.9% `Member`, 0.05% `Insert`, 0.05% `Delete`.

	Number of Threads			
Implementation	**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

-

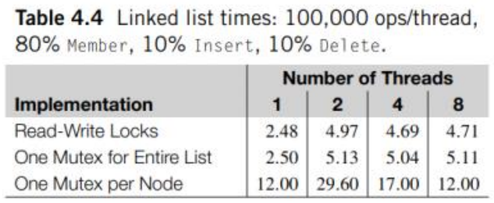


Table 4.4 Linked list times: 100,000 ops/thread, 80% `Member`, 10% `Insert`, 10% `Delete`.

	Number of Threads			
Implementation	**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

- The best solution is the read write locks in most cases