

Here, `thread` is the handle to the thread to be canceled. A thread may cancel itself or cancel other threads. When a call to this function is made, a cancellation is sent to the specified thread. It is not guaranteed that the specified thread will receive or act on the cancellation. Threads can protect themselves against cancellation. When a cancellation is actually performed, cleanup functions are invoked for reclaiming the thread data structures. After this the thread is canceled. This process is similar to termination of a thread using the `pthread_exit` call. This is performed independently of the thread that made the original request for cancellation. The `pthread_cancel` function returns after a cancellation has been sent. The cancellation may itself be performed later. The function returns a 0 on successful completion. This does not imply that the requested thread has been canceled; it implies that the specified thread is a valid thread for cancellation.

7.8 Composite Synchronization Constructs

While the Pthreads API provides a basic set of synchronization constructs, often, there is a need for higher level constructs. These higher level constructs can be built using basic synchronization constructs. In this section, we look at some of these constructs along with their performance aspects and applications.

7.8.1 Read-Write Locks

In many applications, a data structure is read frequently but written infrequently. For such scenarios, it is useful to note that multiple reads can proceed without any coherence problems. However, writes must be serialized. This points to an alternate structure called a read-write lock. A thread reading a shared data item acquires a read lock on the variable. A read lock is granted when there are other threads that may already have read locks. If there is a write lock on the data (or if there are queued write locks), the thread performs a condition wait. Similarly, if there are multiple threads requesting a write lock, they must perform a condition wait. Using this principle, we design functions for read locks `mylib_rwlock_rlock`, write locks `mylib_rwlock_wlock`, and unlocking `mylib_rwlock_unlock`.

The read-write locks illustrated are based on a data structure called `mylib_rwlock_t`. This structure maintains a count of the number of readers, the writer (a 0/1 integer specifying whether a writer is present), a condition variable `readers_proceed` that is signaled when readers can proceed, a condition variable `writer_proceed` that is signaled when one of the writers can proceed, a count `pending_writers` of pending writers, and a mutex `read_write_lock` associated with the shared data structure. The function `mylib_rwlock_init` is used to initialize various components of this data structure.

The function `mylib_rwlock_rlock` attempts a read lock on the data structure. It checks to see if there is a write lock or pending writers. If so, it performs a condition wait on the condition variable `readers_proceed`, otherwise it increments the count of

readers and performs a write lock on the variable `writer_proceed`.

The function `mylib_rwlock_wlock` there is a write lock on the data. If there are readers left and there are pending writers, `writer_proceed` signals all the readers to proceed as follows:

```

1  typedef struct mylib_rwlock_t {
2      int readers;
3      int writer;
4      pthread_t *readers;
5      pthread_t *writers;
6      int pending_writers;
7      pthread_mutex_t read_write_lock;
8  } mylib_rwlock_t;
9
10
11 void mylib_rwlock_rlock(mylib_rwlock_t *rwlock, pthread_t *thread) {
12     pthread_mutex_lock(&rwlock->read_write_lock);
13     pthread_t *readers;
14     pthread_t *writers;
15     pthread_mutex_unlock(&rwlock->read_write_lock);
16 }
17
18 void mylib_rwlock_wlock(mylib_rwlock_t *rwlock, pthread_t *thread) {
19     /* if there are readers, wait for them to finish */
20     wait_for_readers(rwlock);
21
22     pthread_mutex_lock(&rwlock->read_write_lock);
23     while ((rwlock->readers != 0) || (rwlock->writer != 0))
24         pthread_mutex_unlock(&rwlock->read_write_lock);
25     rwlock->writer = 1;
26     pthread_mutex_unlock(&rwlock->read_write_lock);
27 }
28
29
30
31 void mylib_rwlock_unlock(mylib_rwlock_t *rwlock) {
32     /* if there are pending writers, signal them to proceed */
33     pthread_mutex_lock(&rwlock->read_write_lock);
34     rwlock->pending_writers--;
35     pthread_mutex_unlock(&rwlock->read_write_lock);
36
37     pthread_mutex_lock(&rwlock->read_write_lock);
38     while ((rwlock->readers != 0) || (rwlock->writer != 0))
39         pthread_mutex_unlock(&rwlock->read_write_lock);
40     pthread_mutex_unlock(&rwlock->read_write_lock);
41 }

```

readers and proceeds to grant a read lock. The function `mylib_rwlock_wlock` attempts a write lock on the data structure. It checks to see if there are readers or writers; if so, it increments the count of pending writers and performs a condition wait on the condition variable `writer_proceed`. If there are no readers or writer, it grants a write lock and proceeds.

The function `mylib_rwlock_unlock` unlocks a read or write lock. It checks to see if there is a write lock, and if so, it unlocks the data structure by setting the `writer` field to 0. If there are readers, it decrements the number of readers `readers`. If there are no readers left and there are pending writers, it signals one of the writers to proceed (by signaling `writer_proceed`). If there are no pending writers but there are pending readers, it signals all the reader threads to proceed. The code for initializing and locking/unlocking is as follows:

```

1  typedef struct {
2      int readers;
3      int writer;
4      pthread_cond_t readers_proceed;
5      pthread_cond_t writer_proceed;
6      int pending_writers;
7      pthread_mutex_t read_write_lock;
8  } mylib_rwlock_t;
9
10
11 void mylib_rwlock_init (mylib_rwlock_t *l) {
12     l -> readers = 1 -> writer = 1 -> pending_writers = 0;
13     pthread_mutex_init(&(l -> read_write_lock), NULL);
14     pthread_cond_init(&(l -> readers_proceed), NULL);
15     pthread_cond_init(&(l -> writer_proceed), NULL);
16 }
17
18 void mylib_rwlock_rlock(mylib_rwlock_t *l) {
19     /* if there is a write lock or pending writers, perform condition
20     wait.. else increment count of readers and grant read lock */
21
22     pthread_mutex_lock(&(l -> read_write_lock));
23     while ((l -> pending_writers > 0) || (l -> writer > 0))
24         pthread_cond_wait(&(l -> readers_proceed),
25             &(l -> read_write_lock));
26     l -> readers ++;
27     pthread_mutex_unlock(&(l -> read_write_lock));
28 }
29
30
31 void mylib_rwlock_wlock(mylib_rwlock_t *l) {
32     /* if there are readers or writers, increment pending writers
33     count and wait. On being woken, decrement pending writers
34     count and increment writer count */
35
36     pthread_mutex_lock(&(l -> read_write_lock));
37     while ((l -> writer > 0) || (l -> readers > 0)) {
38         l -> pending_writers ++;
39         pthread_cond_wait(&(l -> writer_proceed),
40             &(l -> read_write_lock));
41     }

```

```

42 l -> pending_writers --;
43 l -> writer ++
44 pthread_mutex_unlock(&(l -> read_write_lock));
45 }
46
47
48 void mylib_rwlock_unlock(mylib_rwlock_t *l) {
49     /* if there is a write lock then unlock, else if there are
50     read locks, decrement count of read locks. If the count
51     is 0 and there is a pending writer, let it through, else
52     if there are pending readers, let them all go through */
53
54     pthread_mutex_lock(&(l -> read_write_lock));
55     if (l -> writer > 0)
56         l -> writer = 0;
57     else if (l -> readers > 0)
58         l -> readers --;
59     pthread_mutex_unlock(&(l -> read_write_lock));
60     if ((l -> readers == 0) && (l -> pending_writers > 0))
61         pthread_cond_signal(&(l -> writer_proceed));
62     else if (l -> readers > 0)
63         pthread_cond_broadcast(&(l -> readers_proceed));
64 }

```

We now illustrate the use of read-write locks with some examples.

Example 7.7 Using read-write locks for computing the minimum of a list of numbers

A simple use of read-write locks is in computing the minimum of a list of numbers. In our earlier implementation, we associated a lock with the minimum value. Each thread locked this object and updated the minimum value, if necessary. In general, the number of times the value is examined is greater than the number of times it is updated. Therefore, it is beneficial to allow multiple reads using a read lock and write after a write lock only if needed. The corresponding program segment is as follows:

```

1 void *find_min_rw(void *list_ptr) {
2     int *partial_list_pointer, my_min, i;
3     my_min = MIN_INT;
4     partial_list_pointer = (int *) list_ptr;
5     for (i = 0; i < partial_list_size; i++)
6         if (partial_list_pointer[i] < my_min)
7             my_min = partial_list_pointer[i];
8     /* lock the mutex associated with minimum_value and
9     update the variable as required */
10    mylib_rwlock_rlock(&read_write_lock);
11    if (my_min < minimum_value) {
12        mylib_rwlock_unlock(&read_write_lock);
13        mylib_rwlock_wlock(&read_write_lock);
14        minimum_value = my_min;
15    }
16    /* and unlock the mutex */
17    mylib_rwlock_unlock(&read_write_lock);

```

```

18    pthread_exit(0);
19 }

```

Programming Notes In t
ement in its partial list. It th
the global minimum value. If
minimum value (thus requiring
lock is sought. Once the write
updated. The performance gai
number of threads and the nun
case when the first value of the
write locks are subsequently se
performs better. In contrast, if e
locks are superfluous and add c

Example 7.8 Using read-

A commonly used operation i
space search is the search of a k
table. In our example, we assu
entries into linked lists. Each li
lists are not being updated and
of this program: one using mut
this section.

The mutex lock version
the mutex associated with the t
linked list. The thread function

```

1 manipulate_hash_table(i
2     int table_index, fo
3     struct list_entry *
4
5     table_index = hash(
6     pthread_mutex_lock(
7     found = 0;
8     node = hash_table[ta
9     while ((node != NULL
10        if (node -> valu
11            found = 1;
12        else
13            node = node
14    }
15    pthread_mutex_unlock
16    if (found)
17        return(1);
18    else
19        insert_into_hash
20 }

```

```

18     pthread_exit(0);
19 }

```

Programming Notes In this example, each thread computes the minimum element in its partial list. It then attempts a read lock on the lock associated with the global minimum value. If the global minimum value is greater than the locally minimum value (thus requiring an update), the read lock is relinquished and a write lock is sought. Once the write lock has been obtained, the global minimum can be updated. The performance gain obtained from read-write locks is influenced by the number of threads and the number of updates (write locks) required. In the extreme case when the first value of the global minimum is also the true minimum value, no write locks are subsequently sought. In this case, the version using read-write locks performs better. In contrast, if each thread must update the global minimum, the read locks are superfluous and add overhead to the program. ■

Example 7.8 Using read-write locks for implementing hash tables

A commonly used operation in applications ranging from database query to state space search is the search of a key in a database. The database is organized as a hash table. In our example, we assume that collisions are handled by chaining colliding entries into linked lists. Each list has a lock associated with it. This lock ensures that lists are not being updated and searched at the same time. We consider two versions of this program: one using mutex locks and one using read-write locks developed in this section.

The mutex lock version of the program hashes the key into the table, locks the mutex associated with the table index, and proceeds to search/update within the linked list. The thread function for doing this is as follows:

```

1  manipulate_hash_table(int entry) {
2      int table_index, found;
3      struct list_entry *node, *new_node;
4
5      table_index = hash(entry);
6      pthread_mutex_lock(&hash_table[table_index].list_lock);
7      found = 0;
8      node = hash_table[table_index].next;
9      while ((node != NULL) && (!found)) {
10         if (node->value == entry)
11             found = 1;
12         else
13             node = node->next;
14     }
15     pthread_mutex_unlock(&hash_table[table_index].list_lock);
16     if (found)
17         return(1);
18     else
19         insert_into_hash_table(entry);
20 }

```