

CS 292

Introduction to Parallel Computing

Spring 2008

VANDERBILT



School of Engineering

Announcements

- Read Sections 7.6-7.8
- Programming assignment #1 has been posted to Oak
 - Two programs using Pthreads
 - Due Friday, 2/29/08
- Questions??

Chapter 7 Topic Overview

Shared memory programming:

- Thread Basics ✓
- The POSIX Thread API ✓
- Synchronization Primitives in Pthreads ✓
- Controlling Thread and Synchronization Attributes
- Composite Synchronization Constructs
- OpenMP: a Standard for Directive Based Parallel Programming

Controlling Thread and Synchronization Attributes

- In the discussion of Pthreads up to this point, we have seen that the APIs for creating a thread, mutex, or condition variable take an attribute as an argument.
- The Pthreads API allows a programmer to change the default attributes of entities using *attributes objects*.
- An attributes object is a data-structure that describes entity (thread, mutex, condition variable) properties.
- Once these properties are set, the attributes object can be passed to the method initializing the entity.
- Enhances modularity, readability, and ease of modification.

Attributes Objects for Threads

- Use `pthread_attr_init` to create an attributes object for threads, which initializes the attributes object with default values.
- Individual properties associated with the attributes object can be changed using the following functions:
`pthread_attr_setdetachstate,`
`pthread_attr_setguardsize_np,`
`pthread_attr_setstacksize,`
`pthread_attr_setinheritsched,`
`pthread_attr_setschedpolicy,` **and**
`pthread_attr_setschedparam`

Attributes Objects for Mutexes

- You initialize the attributes object for mutexes using the function: `pthread_mutexattr_init`.
- The function `pthread_mutexattr_settype_np` can be used for setting the type of mutex specified by the mutex attributes object:

```
pthread_mutexattr_settype_np (  
    pthread_mutexattr_t *attr,  
    int type);
```
- Here, `type` specifies the type of the mutex and can take one of:
 - `PTHREAD_MUTEX_NORMAL_NP`
 - `PTHREAD_MUTEX_RECURSIVE_NP`
 - `PTHREAD_MUTEX_ERRORCHECK_NP`

Types of Mutexes

- Pthreads supports three types of mutexes - normal, recursive, and error-check.
- A normal mutex deadlocks if a thread that already has a lock tries a second lock on it.
- A recursive mutex allows a single thread to lock a mutex as many times as it wants. It simply increments a count on the number of locks. Each unlock decrements the count. A lock is relinquished by a thread when the count becomes zero. Useful for recursive thread functions.
- An error check mutex reports an error when a thread with a lock tries to lock it again (as opposed to deadlocking in the first case, or granting the lock, as in the second case).

Types of Mutexes

- Consider this case with a normal mutex and with a recursive mutex:

```
search_tree(void *tree_ptr)
{
    struct node *node_ptr = (struct node*) tree_ptr;
    pthread_mutex_lock(&tree_lock);
    if (is_desired_node(node_ptr)) {
        print_node(node_ptr);
        found = true; // node found
    } else {
        if (node_ptr->left != NULL)
            found = search_tree((void*) node_ptr->left);
        if (!found && node_ptr->right != NULL)
            found = search_tree((void*) node_ptr->right);
    }
    pthread_mutex_unlock(&tree_lock);
    return found;
}
```

Composite Synchronization Constructs

- By design, Pthreads provide support for a basic set of operations.
- Higher level constructs can be built using basic synchronization constructs.
- We discuss two such constructs - *read-write locks* and *barriers*.

Read-Write Locks

- In many applications, a data structure is read frequently but written infrequently. For such applications, we should use read-write locks.
 - This allows multiple reader threads to access the shared data structure simultaneously while serializing threads that write.
- 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.
- If there are multiple threads requesting a write lock, they must perform a condition wait.
- With this description, we can design functions for read locks `mylib_rwlock_rlock`, write locks `mylib_rwlock_wlock`, and unlocking `mylib_rwlock_unlock`.

Read-Write Locks

- The lock data type `mylib_rwlock_t` holds the following:
 - 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

Read-Write Locks

```
typedef struct {
    int readers;
    int writer;
    pthread_cond_t readers_proceed;
    pthread_cond_t writer_proceed;
    int pending_writers;
    pthread_mutex_t read_write_lock;
} mylib_rwlock_t;
```

```
void mylib_rwlock_init (mylib_rwlock_t *lock) {
    lock->readers = lock->writer = lock->pending_writers = 0;
    pthread_mutex_init(&(lock->read_write_lock), NULL);
    pthread_cond_init(&(lock->readers_proceed), NULL);
    pthread_cond_init(&(lock->writer_proceed), NULL);
}
```

Read-Write Locks

```
void mylib_rwlock_rlock(mylib_rwlock_t *lock) {
    /* get a read lock.
       if there is a write lock or pending writers, perform
       condition wait.. else increment count of readers and
       grant read lock */
    pthread_mutex_lock(&(lock->read_write_lock));
    while ((lock->writer > 0) || (lock->pending_writers > 0))
        pthread_cond_wait(&(lock->readers_proceed),
                          &(lock->read_write_lock));
    lock->readers++;
    pthread_mutex_unlock(&(lock->read_write_lock));
}
```

Read-Write Locks

```
void mylib_rwlock_wlock(mylib_rwlock_t *lock) {
    /* get a write lock.
       if there are readers or writers, increment pending writers
       count and wait. On being woken, decrement pending writers
       count and increment writer count */

    pthread_mutex_lock(&(lock->read_write_lock));
    while ((lock->writer > 0) || (lock->readers > 0)) {
        lock->pending_writers++;
        pthread_cond_wait(&(lock->writer_proceed),
                        &(lock->read_write_lock));
    }
    lock->pending_writers--; /* does this belong in the loop? */
    lock->writer++;
    pthread_mutex_unlock(&(lock->read_write_lock));
}
```

Read-Write Locks

```
void mylib_rwlock_unlock(mylib_rwlock_t *lock) {
/* if there is a write lock then unlock, else if there are
   read locks, decrement count of read locks. If the count is
   0 and there is a pending writer, let it through, else if
   there are pending readers, let them all go through */
pthread_mutex_lock(&(lock->read_write_lock));
if (lock->writer > 0)
    lock->writer = 0;
else if (lock->readers > 0)
    lock->readers--;
pthread_mutex_unlock(&(lock->read_write_lock));
if ((lock->readers == 0) && (lock->pending_writers > 0))
    pthread_cond_signal(&(lock->writer_proceed));
else if (lock->readers > 0) /* is this final IF correct? */
    pthread_cond_broadcast(&(lock->readers_proceed));
}
```

Is it a problem that we release the lock on the shared data structure and then continue to read it's values and perform signals based on those values??

Read-Write Locks

- Let's rewrite our best cost computation to use read-write locks which allows multiple threads to simultaneously read the best cost computed so far:

```
void *find_best_cost(void *list_ptr) {
    ....
    mylib_rwlock_rlock(&read_write_lock);
    if (my_cost < best_cost) {
        mylib_rwlock_unlock(&read_write_lock);
        mylib_rwlock_wlock(&read_write_lock);
        best_cost = my_cost;  /* RACE CONDITION?? */
    }
    mylib_rwlock_unlock(&read_write_lock);
    ....
}
```