

CS 292

Introduction to Parallel Computing

Spring 2008

VANDERBILT



School of Engineering

Announcements

- Read Section 7.5
- 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

Thread Review: Creation and Termination

```
#include <pthread.h>
```

```
int pthread_create (  
    pthread_t *thread_handle,  
    const pthread_attr_t *attribute,  
    void * (*thread_function)(void *),  
    void *arg);
```

```
int pthread_join (  
    pthread_t thread_handle,  
    void **ptr);
```

Synchronization Primitives in Pthreads

- Threads communicate with one another by accessing shared data.
- When multiple threads attempt to manipulate the same data item, the results can often be incoherent.
 - This is a result of the fact that the reads/writes performed by one thread may be interleaved with those of another thread
- Proper care must be taken to synchronize the accesses, particularly if one or more threads is attempting to update the shared data.

Synchronization Primitives in Pthreads

- Consider:

```
/* each thread tries to update the shared
   variable best_cost as follows */
if (my_cost < best_cost)
    best_cost = my_cost;
```

- Assume that there are two threads, the initial value of `best_cost` is 100, and the values of `my_cost` are 50 and 75 for threads `t1` and `t2`, respectively.
- Depending on the schedule of the threads, the value of `best_cost` could be 50 or 75! And worse, the value 75 does not correspond to any serialization of the threads.
- This is known as a *race condition*.

Mutual Exclusion

- The code in the previous example corresponds to a critical segment; i.e., a segment that must be executed by only one thread at any time.
- Critical segments in Pthreads are implemented using mutual exclusion locks, or *mutex-locks*.
- Mutex-locks have two states: locked and unlocked. At any point of time, only one thread can lock a mutex-lock.
- A thread entering a critical segment first tries to get a lock. It goes ahead when the lock is granted.

Mutual Exclusion

- The use of a mutex-lock follows these general guidelines:
 1. Declare an object of type `pthread_mutex_t`
 2. Initialize the object by calling `pthread_mutex_init()`
 3. Call `pthread_mutex_lock()` to gain exclusive access
 4. Call `pthread_mutex_unlock()` to release the exclusive access
 5. Call `pthread_mutex_destroy()` to get rid of the object when it is no longer needed

Mutual Exclusion

- Pthreads API:

```
int pthread_mutex_init (pthread_mutex_t *mutex_lock,  
    const pthread_mutexattr_t *lock_attr);
```

```
int pthread_mutex_lock (pthread_mutex_t *mutex_lock);
```

```
int pthread_mutex_unlock (pthread_mutex_t *mutex_lock);
```

Mutual Exclusion

- A call to `pthread_mutex_lock()` attempts to lock the mutex-lock given as an argument.
- If the mutex-lock is already locked, then the calling thread blocks.
- If the mutex-lock is available, the call will lock it and execution of the thread continues.
- The lock operation is considered *atomic*; that is, it is completed as if a single instruction. Thus it is impossible for a lock to be given to two threads simultaneously.

Mutual Exclusion

- We can now write our previously incorrect code segment as:

```
pthread_mutex_t best_cost_lock;
...
main() {
    ....
    pthread_mutex_init(&best_cost_lock, NULL);
    ....
}

void *find_best_cost(void *list_ptr) {
    ....
    pthread_mutex_lock(&best_cost_lock);
    if (my_cost < best_cost)
        best_cost = my_cost;
    pthread_mutex_unlock(&best_cost_lock);
}
```

Mutual Exclusion

- Be sure to observe these points:
 1. No thread should attempt to lock/unlock a mutex-lock that has not been initialized
 2. The thread that locks a mutex-lock must be the thread that unlocks it
 3. No thread should have the mutex-lock locked when it is destroyed
 4. Good practice also says that any mutex-lock that is initialized should eventually be destroyed

Producer-Consumer Programming

- A common use of thread programming is to handle producer-consumer relationships.
- Producers create data and add it to a work queue.
- Consumers pick up data from the work queue and process it.
- Data creation and processing may be independent in which case they can be executed simultaneously.
 - The work queue/buffer is a shared data structure and thus access must be synchronized

Producer-Consumer Using Locks

- A multi-threaded producer-consumer scenario imposes the following constraints:
- The producer thread must not overwrite the shared buffer when the previous data has not been picked up by a consumer thread.
- The consumer threads must not pick up data until there is something present in the shared buffer.
- Individual consumer threads should pick up tasks one at a time.

Producer-Consumer Using Locks

```
int task_available;           // shared data
pthread_mutex_t task_queue_lock; // lock for shared data
...
main() {
    ....
    task_available = 0;
    pthread_mutex_init(&task_queue_lock, NULL);
    ....
}
void *producer(void *producer_thread_data) {
    ....
    while (!done()) {
        inserted = 0;
        create_task(&my_task);
        while (inserted == 0) {
            pthread_mutex_lock(&task_queue_lock);
            if (task_available == 0) {
                insert_into_queue(my_task);
                task_available = 1;
                inserted = 1;
            }
            pthread_mutex_unlock(&task_queue_lock);
        }
    }
}
```

Producer-Consumer Using Locks

```
void *consumer(void *consumer_thread_data) {
    int extracted;
    struct task my_task;
    /* local data structure declarations */
    while (!done()) {
        extracted = 0;
        while (extracted == 0) {
            pthread_mutex_lock(&task_queue_lock);
            if (task_available == 1) {
                extract_from_queue(&my_task);
                task_available = 0;
                extracted = 1;
            }
            pthread_mutex_unlock(&task_queue_lock);
        }
        process_task(my_task);
    }
}
```

Overheads of Locking

- Locks represent serialization points since critical sections must be executed by only one thread at a time.
- Encapsulating large segments of the program within locks can lead to significant performance degradation.
- It is often possible to reduce the idling overhead associated with locks using an alternate function, `pthread_mutex_trylock`.
- `pthread_mutex_trylock` is typically much faster than `pthread_mutex_lock` on typical systems since it does not have to deal with queues associated with locks for multiple threads waiting on the lock.

Overheads of Locking

- `pthread_mutex_trylock` attempts to obtain the lock.
- If successful, the function returns zero.
- If unsuccessful, the value `EBUSY` is returned.
 - This allows the thread to continue doing more work, rather than block while waiting for the lock.
 - We can then try to obtain the lock again later.

Alleviating Locking Overhead (Example)

```
/* Finding k matches in a list using lock */
void *find_entries(void *start_pointer) {
    /* This is the thread function */
    struct database_record *next_record;
    int count;
    current_pointer = start_pointer;
    do {
        next_record = find_next_entry(current_pointer);
        count = output_record(next_record);
    } while (count < requested_number_of_records);
}
int output_record(struct database_record *record_ptr) {
    int count;
    pthread_mutex_lock(&output_count_lock);
    output_count ++;
    count = output_count;
    pthread_mutex_unlock(&output_count_lock);
    if (count <= requested_number_of_records)
        print_record(record_ptr);
    return (count);
}
```

Alleviating Locking Overhead (Example)

```
/* rewritten output_record function using trylock */
int output_record(struct database_record *record_ptr) {
    int count;
    int lock_status;
    lock_status=pthread_mutex_trylock(&output_count_lock);
    if (lock_status == EBUSY) {
        insert_into_local_list(record_ptr);
        return(0);
    }
    else {
        count = output_count;
        output_count += number_on_local_list + 1;
        pthread_mutex_unlock(&output_count_lock);
        print_records(record_ptr, local_list,
            requested_number_of_records - count);
        return(count + number_on_local_list + 1);
    }
}
```

Condition Variables for Synchronization

- In the earlier producer-consumer example, a thread has to continually obtain a lock before even testing if the work buffer has room for more (or has any data to consume).
- A better solution would be to suspend execution until more room is available (or data has been produced).
- A *condition variable* allows a thread to block itself until a specified predicate becomes true.

Condition Variables for Synchronization

- A condition variable is associated with the predicate. When the predicate becomes true, the condition variable is used to signal one or more threads waiting on the condition.
- A condition variable always has a mutex associated with it. A thread locks this mutex and tests the predicate defined on the shared variable.
- If the predicate is not true, the thread waits on the condition variable associated with the predicate using the function `pthread_cond_wait`.
 - This also releases the lock on the mutex so that others can change the condition variable
- At a later time when another thread makes the predicate true, it calls `pthread_cond_signal` to unblock the waiting thread.
 - The signaled (waiting) thread now also has the lock on the mutex

Condition Variables for Synchronization

- Pthreads provides the following functions for condition variables:

```
int pthread_cond_init(pthread_cond_t *cond,  
    const pthread_condattr_t *attr);
```

```
int pthread_cond_wait(pthread_cond_t *cond,  
    pthread_mutex_t *mutex);
```

```
int pthread_cond_signal(pthread_cond_t *cond);
```

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

Producer-Consumer Using Condition Variables

```
int task_available;
pthread_mutex_t task_queue_cond_lock;
pthread_cond_t cond_queue_empty, cond_queue_full;
/* other data structures here */
main() {
    /* declarations and initializations */
    task_available = 0;
    pthread_init();
    pthread_cond_init(&cond_queue_empty, NULL);
    pthread_cond_init(&cond_queue_full, NULL);
    pthread_mutex_init(&task_queue_cond_lock, NULL);

    /* create and join producer and consumer threads */
}
```

Producer-Consumer Using Condition Variables

```
void *producer(void *producer_thread_data) {
    int inserted;
    while (!done()) {
        create_task();
        pthread_mutex_lock(&task_queue_cond_lock);
        while (task_available == 1)
            pthread_cond_wait(&cond_queue_empty,
                              &task_queue_cond_lock);
        insert_into_queue();
        task_available = 1;
        pthread_cond_signal(&cond_queue_full);
        pthread_mutex_unlock(&task_queue_cond_lock);
    }
}
```

Producer-Consumer Using Condition Variables

```
void *consumer(void *consumer_thread_data) {
    while (!done()) {
        pthread_mutex_lock(&task_queue_cond_lock);
        while (task_available == 0)
            pthread_cond_wait(&cond_queue_full,
                              &task_queue_cond_lock);
        my_task = extract_from_queue();
        task_available = 0;
        pthread_cond_signal(&cond_queue_empty);
        pthread_mutex_unlock(&task_queue_cond_lock);
        process_task(my_task);
    }
}
```