Introduction to parallel computing

Shared Memory Programming with Pthreads (2)

Zhiao Shi (modifications by Will French)
Advanced Computing Center for Education & Research
Vanderbilt University
Fork-join model for executing threads in an application
Last time: Thread creation and termination

```c
int pthread_create (  
    pthread_t* thread_p,  
    const pthread_attr_t* attr_p,  
    void* (*start_routine ) (void),  
    void* arg_p)

int pthread_join (  
    pthread_t thread_handle,  
    void **ptr)

void pthread_exit_exit( (void * status)```
Shared vs. Non-Shared (Local) Variables

• Anything declared within a worker thread function is local to each thread (each thread gets its own independent copy).

• Any variable declared with global scope is shared by all threads. E.g.:

```c
int sharedVar; // global scope, shared by all threads
void * threadFunc( void * args ) {
    int localVar; // each thread get its own copy
}
int main()
{
    ...
    sharedVar = 17; // initializing value here, which will be shared across all threads
    ...
    pthread_create(&thread[i], NULL, threadFunc, (void *) &funcArgs[i]); // fork threads
    ...
}
```
Synchronization in Pthreads

• Threads indirectly communicate with one another by accessing shared data.
• When multiple threads attempt to modify 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 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.
## Race Conditions

**Initial balance = $1000**
**Final balance = ?**

<table>
<thead>
<tr>
<th>Time</th>
<th>Withdrawal</th>
<th>Deposit</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_0$</td>
<td>Load (balance = $1000)</td>
<td></td>
</tr>
<tr>
<td>$T_1$</td>
<td>Subtract $100</td>
<td>Load (balance = $1000)</td>
</tr>
<tr>
<td>$T_2$</td>
<td>Store (balance = $900)</td>
<td>Add $100</td>
</tr>
<tr>
<td>$T_3$</td>
<td></td>
<td>Store (balance = $1100)</td>
</tr>
</tbody>
</table>
When is synchronization needed?

• If a shared variable is being modified!
• It’s okay for multiple threads to access a shared variable simultaneously, so long as the threads are not modifying said variable.
• We need a mechanism that allows shared variables to be modified safely....
Mutual Exclusion

- **Critical section**: a code segment that must be executed by only one thread at any time.
- Critical sections 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 section 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:
  
  • Declare an object of type `pthread_mutex_t`
  
  • Initialize the object by calling `pthread_mutex_init()`
  
  • Call `pthread_mutex_lock()` to gain exclusive access
  
  • Call `pthread_mutex_unlock()` to release the exclusive access
  
  • Call `pthread_mutex_destroy()` to get rid of the object when it is no longer needed
Mutual Exclusion

• Pthreads API:

```c
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);

int pthread_mutex_destroy (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:

```c
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:
  • No thread should attempt to lock/unlock a mutex-lock that has not been initialized.
    — Thus initialization must occur before threads calls lock/unlock.
  • The thread that locks a mutex-lock must be the thread that unlocks it.
  • No thread should have the mutex-lock locked when it is destroyed.
  • 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.
• The work queue is shared data and thus access must be synchronized.
  • Data creation and processing may be independent, in which case threads can execute these tasks simultaneously.
Producer-Consumer Using Lock

- A multi-threaded producer-consumer scenario imposes the following constraints:
  - Generally one producer and multiple consumers.
  - The producer thread must not overwrite the shared memory space 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 memory space.
int task_available; // shared data
pthread_mutex_t task_queue_lock; // lock for shared data
int inserted, extracted;

main() {
    ..... 
    task_available = 0; // initialize queue to empty
    pthread_mutex_init(&task_queue_lock, NULL);
    ..... /* create & join producer and consumer threads */ ...
}

void *producer(void *producer_thread_data) {
    ..... 
    while (!done()) {
        create_task(&my_task);
        inserted = 0;
        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);
        }
    }
}
void *consumer(void *consumer_thread_data) {
    ....
    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);
    }
}
Overhead 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`. 
Overhead of Locking

• `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.

```c
int pthread_mutex_trylock(pthread_mutex_t *mutex_lock);
```
Overhead 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.
Next time

• Barrier
• Conditional variable