

CSC 262

Homework 6

Due Nov. 13, 2008

For this homework assignment, you will be asked to reason about a simple concurrent data structure, an atomic array. This is an array of `void*`, meant to be accessed and modified in a thread-safe fashion by multiple threads. There are initially three operations on the array: `get`, `set`, and `getAndSet`. `get` reads a value from the array at a specified index. `set` sets a value in the array at a specified index, and lastly `getAndSet` returns the old value at a specified index and sets it to a new value. The initial (NOT thread safe) definition of the atomic array and its functions is:

```
typedef struct{
    void* array[ARRAY_LENGTH];
    mutex* array_lock;
} AtomicArray;

void* get(AtomicArray* arr, int idx){
    return arr->array[idx];
}

void set(AtomicArray* arr, int idx, void* val){
    arr->array[idx] = val;
}

void* getAndSet(AtomicArray* arr, int idx, void* val){
    void* old = arr->array[idx];
    arr->array[idx] = val;
    return old;
}
```

For this assignment assume that there are two functions for manipulating mutexes. `lock(mutex* lck)` acquires `lck` if it is unlocked, otherwise the thread is blocked until `lck` is released by the holding thread. `unlock(mutex* lck)` releases the held lock.

1. Assume that there are 3 threads (T1 through T3). If T1 is executing `get(arr, 4)`, T2 is executing `set(arr, 7, (void*)42)`, and T3 is executing `getAndSet(arr, 7, (void*)12)`; is there an interleaving where these threads will interfere? (assume that `arr` is initialized to all NULL values).
2. Identify the critical sections in each of the methods and surround them with appropriate `lock` and `unlock` calls.
3. Recall that locks are a pessimistic form of concurrency control. Consider the case where you have two threads, T1 and T2. If T1 is executing `get(arr, 4)` and T2 is executing

`set(arr, 5, (void*)42);` do these threads interfere (i.e. are locks necessary in this case)?

4. Because many operations on a large array may not interfere, a common practice is to replace the large whole-array lock with an array of so-called “fine-grained” locks. In this case, there is an array of locks as large as the atomic array itself. Each lock protects a specific element (the one at the same index). In this case, only if two threads are attempting to access the same element will one of them be blocked, which is important in situations where there may be hundreds of threads all trying to access this shared array. Re-implement the atomic array structure and the `get`, `set`, and `getAndSet` functions to work with an array of locks rather than just one large lock for the whole array.