Project: Work-Stealing Algorithm

In this part of the assignment, we will extend the deque data structure from Part 1 of the assignment to implement a work-stealing algorithm. A work-stealing algorithm is primarily used for balancing work across threads. A set of task threads are assigned individual tasks to execute. Each task generally consists of smaller basic units of work, for example, in our first assignment, each task consisted of units of work that involved writing thread-ids to the individual deque slots. Thus the smallest piece of work was to write the thread-id in a single deque slot. With a work-stealing algorithm, task threads that finish their task can steal basic units of unfinished work or entire unfinished tasks from other threads.

Your implementation of this work-stealing algorithm will be based on:

Thread Scheduling for Multiprogrammed Multiprocessors by Arora et al [1], available online at:
http://www.cs.berkeley.edu/~nimar/papers/workstealing-tocs-01.pdf

This algorithm is non-blocking in nature, and will require familiarization with atomic primitives such as CAS [2] (also described in Section 9.1 of Solihin). Section 3.3 of the paper is most relevant for this part of the project. The three non-blocking functions listed in Figure 5 in the paper need to be implemented.

Implementation

In this project, the focus is on the implementation of the work-stealing algorithm. The tasks performed by the individual threads are quite simple, with the goal being to demonstrate a non-blocking deque. Your program will consist of the framework from the previous part of the assignment. Each task-thread deque is implemented according to the deque implementation from the paper [1].

1. Each task thread has a local double-ended queue, or deque. This data structure can be implemented as an array with two index variables, viz., *top* and *bottom*, with both initialized to zero. The deque is a fixed-sized array with 2048 entries.
2. The manager "enqueues" the task for each task thread by initializing each task thread's task queue, i.e., by resetting the array *bottom* index and the array entries.
    a. The array *bottom* index is set to 2048
    b. The array *top* index is set to 0.
    c. The array entries are initialized to either zeroes or a sentinel value. The sentinel value in our case will be −1, and every 256th entry in the array should be initialized to the sentinel value of −1. So in an array of 2048 total entries, you would end up with 8 such sentinel entries.
3. The manager then signals the task threads to start executing the tasks, and waits for them to finish the tasks.

4. The task for each thread is to write its thread ID [4] into each of the deque slots it claims.
5. The task thread starts from the *bottom* of the array which is initialized to 2048.
6. For each slot, the task thread reads the value in the current slot and tries to claim the slot using the `popBottom` function in Figure 5 of the paper [1], if the value in the slot is zero.
7. The task thread pushes work onto the deque, if the value in the slot is the sentinel value by writing a zero value in the slot using the `pushBottom` function in figure 5 of the paper[1].
8. For each slot claimed, it writes its thread-id into the slot.
9. Once the thread finishes its deque, it tries to steal slots from other threads.
10. The task thread picks a thread from which to steal a unit of work. Implement it in a reasonable way; this could be using round-robin style or by a random-number generator modulo the number of threads.
11. Once the task thread chooses a thread to steal from, it uses the `popTop` function in figure 5 of the paper [1] to steal a slot from the deque of the chosen thread. It only steals slots that contain zero values. It aborts the steal if it sees a slot with a non-zero value (either a sentinel or another thread's thread ID) in it, and tries to steal from another thread.
12. The task is terminated once each task thread has made an unsuccessful steal attempt from each of the other task threads. You can chose to implement other reasonable termination conditions.
13. What we have described so far constitutes one batch of tasks.
14. Once all the task threads have finished their assigned tasks in the current batch, the manager can enqueue the next batch of tasks, i.e., return to step 2.
15. Three such batches of tasks need to be performed, with each task thread expected to print out simple statistics such as:
    a. number of steal attempts
    b. number of successful steals
16. The main threads and the task threads should then exit gracefully.
17. The implementation should use the *tag* values concatenated with the *top* index to avoid the ABA problem discussed in Problem Set 5 and the paper.
18. The implementation should also use the compare-and-swap atomic primitive in the implementation of the deque functions. This is available as a gcc built-in function [3].

Finally, answer the following questions:

1. In our implementation, would the ABA issue still exist if work wasn't being pushed back by task threads? Describe why or why not.
2. Is there a need for any explicit memory-fence instructions in any of the steal functions? Describe why or why not.

Submission

The implementation of this assignment requires the use of a machine that supports 64-bit data types. Consequently, you will need to use remote-linux.eos.ncsu.edu to implement your assignment. Submit a file called proj.c that compiles using the following command-line arguments:

```
    gcc -g proj.c -Wall -lpthread -lrt
```

In addition, submit a plain-text file containing the short answers to the two questions in the previous section.

A working implementation would satisfy the functional requirements defined above and be deadlock and livelock free. Non-compiling assignments will receive a zero.

References

1. Thread Scheduling for Multiprogrammed Multiprocessors by Arora et al., http://www.cs.berkeley.edu/~nimar/papers/workstealing-tocs-01.pdf
2. http://en.wikipedia.org/wiki/Compare_and_swap
3. http://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/Atomic-Builtins.html
4. https://computing.llnl.gov/tutorials/pthreads/man/pthread_self.txt