Assignment #3: All Hands on Deques

Lalit Prasad Peri (lalitprasad@vt.edu)
Will Downey (willd20@vt.edu)
(Homework and Programming Assignment Group 15)

Link to github : https://github.com/Virginia-Tech-CS-Systems-Courses/All-Hands-on-Dequeues-group15/tree/main

1. Functional- Tests / Performance measurements:

Test were run with *run-test.sh*, updated to sequentially run all the functional correctness checks and performance checks.

- **Functional Tests on Deque** tries all possible combinations of push/pops from both the ends. There two possible memory allocations tests fixed allocation array and dynamic qnode based allocations, which is tested upto 64 qnodes / 16 threads, which prints PASS/FAIL status with each test.

```
---------------------------------------
Deque - Functional Correctness tests
---------------------------------------
------------------------------
testing options() with no queue
dque_options (DQUEOPT_HEADSIZ):                                    Pass
qhead size: 48
dque_options (DQUEOPT_NODESIZ):                                    Pass
qnode size: 24
------------------------------
dque_push_back:                                                    Pass
dque_push_back:                                                    Pass
dque_push_back:                                                    Pass
dque_push_back:                                                    Pass
dque_push_back:                                                    Pass
dque_front data == front of queue:                                 Pass

remove the five string elements
dque_pop_front:                                                    Pass
dque_pop_front:                                                    Pass
dque_pop_front:                                                    Pass
dque_pop_front:                                                    Pass
dque_pop_front:                                                    Pass
dque_empty:                                                        Pass
remove the five record elements
dque_pop_front:                                                    Pass
dque_pop_front:                                                    Pass
dque_pop_front:                                                    Pass
dque_pop_front:                                                    Pass
dque_pop_front:                                                    Pass
------------------------------
fill queue with five order integer elements using que_push
dque_que_insert:                                                   Pass
dque_que_insert:                                                   Pass
dque_que_insert:                                                   Pass
dque_que_insert:                                                   Pass
dque_que_insert:                                                   Pass
 0 1 2 3 4
dque_que_empty == 0:                                               Pass
dque_que_size  == 5:                                               Pass
dque_que_back  == 4:                                               Pass
dque_que_front == 0:                                               Pass
dque_que_pop   == 0:                                               Pass
dque_que_pop   == 1:                                               Pass
dque_que_pop   == 2:                                               Pass
dque_que_pop   == 3:                                               Pass
dque_que_pop   == 4:                                               Pass
dque_que_empty == 1:                                               Pass

++++++++++++++++++++++++++++++++++++++++
OVERALL DEQUE FUNCTIONAL TEST STATUS:                              PASS
++++++++++++++++++++++++++++++++++++++++
```

- **Locked and Lock-free:** Locked implementation, we use fine grained TTAS & mutex locks invoked in push/pop_front/back functions. Lock-free implementation used atomic compare&set and compare&exchange semantics.

- **Performance Tests**: measured in time of operations in seconds, scaling from 1 to 16 threads. While spawing threads in done using pthreads, aggregation is done using Cilk_sync, hence both cilk_worker and thread_count variables are passed in binaries.
    - ***Fibonacci (upto fib-40)***
    - ***Qsort – 1 Million Integer sorting***
    - ***Chess test (N-Queens probability)***

```
-------------------------------------
 Fibonacci Test (Deque-Lock)
-------------------------------------
fib(40) = 102334155     Time(fib) = 6.768299572 sec
fib(40) = 102334155     Time(fib) = 3.583296251 sec
fib(40) = 102334155     Time(fib) = 1.770456323 sec
fib(40) = 102334155     Time(fib) = 0.968938153 sec
fib(40) = 102334155     Time(fib) = 0.593530352 sec
-------------------------------------
 Qsort Test - 1 Million Number Sorting (Deque Lock)
-------------------------------------
Time(sample_qsort) = 0.140534353 sec
Time(sample_qsort) = 0.080839710 sec
Time(sample_qsort) = 0.051283098 sec
Time(sample_qsort) = 0.032594895 sec
Time(sample_qsort) = 0.047535250 sec
-------------------------------------
 Chess Test - N Queens Problem - 16 (Deque-Lock)
-------------------------------------
Time(nqueens) = 57.259795772 sec
Time(nqueens) = 28.093670772 sec
Time(nqueens) = 14.210697681 sec
Time(nqueens) = 7.474364479 sec
Time(nqueens) = 4.305742085 sec
-------------------------------------
-------------------------------------
-------------------------------------
 Fibonacci Test (Deque-Lock-Free)
-------------------------------------
fib(40) = 102334155     Time(fib) = 6.585685730 sec
fib(40) = 102334155     Time(fib) = 3.634841760 sec
fib(40) = 102334155     Time(fib) = 2.124401727 sec
fib(40) = 102334155     Time(fib) = 1.056822500 sec
fib(40) = 102334155     Time(fib) = 0.651610867 sec
-------------------------------------
 Qsort Test - 1 Million Number Sorting (Deque-Lock-Free)
-------------------------------------
Time(sample_qsort) = 0.148339605 sec
Time(sample_qsort) = 0.080518907 sec
Time(sample_qsort) = 0.051614994 sec
Time(sample_qsort) = 0.031525487 sec
Time(sample_qsort) = 0.035333733 sec
-------------------------------------
 Chess Test - N Queens Problem - 8 (Deque-Lock-Free)
-------------------------------------
Time(nqueens) = 58.234281072 sec
Time(nqueens) = 28.904744897 sec
Time(nqueens) = 14.943900469 sec
Time(nqueens) = 8.331215022 sec
Time(nqueens) = 4.491896053 sec
=========================================
 END OF ALL TESTS
=========================================
```

2. Deque Functions:

- **Push front/back**

```
dque_push_back(                                 /* insert node onto back of queue*/
dque_qhead     *queue,                          /* queue to have node inserted  */
void           *data )                          /* data to insert into queue    */
{                                               /*------------------------------*/
    dque_err        errcode = DQUEERR_NOERR;    /* non-zero indicates failure   */
    dque_qnode     *node;                       /* pointer to inserted node     */

    if (queue == NULL_QUEUE) {                  /* invalid queue pointer?       */
        errcode = DQUEERR_NOQUEUE;
    } else if (data == (void *)NULL) {          /* invalid data pointer?        */
        errcode = DQUEERR_NODATA;
    } else if (getfree( queue ) == NULL_NODE && ((errcode = dque_myalloc( queue )) !=
DQUEERR_NOERR)) {
        ;                   /* no free nodes? try to create more free nodes */
    } else if ((errcode = dque_mydelete( &getfree( queue ), &node, NOROTATE, &getfcnt(
queue ) )) == DQUEERR_NOERR) {
        spin_lock(&sl);                 /* Spinlock-Lock or Atomic or compare & swap*/
        setdata( node, data );          /* got node, set the data       */
        spin_unlock(&sl);               /* Spinlock-unLock or Atomic or compare & swap*/
    }
    if (errcode == DQUEERR_NOERR) {         /* no error? insert node        */
        errcode = dque_myinsert( &gethead( queue ), node, NOROTATE, &gethcnt( queue ) );
    }                                       /* NOROTATE to insert at back   */
    return (errcode);
}
```

- **Pop front/back**

```
dque_pop_front(                                 /* remove first qnode from queue*/
dque_qhead     *queue,                          /* queue to have node removed   */
void          **data )                          /* returned pointer to data     */
{                                               /*------------------------------*/
    dque_err        errcode = DQUEERR_NOERR;/* non-zero indicates failure   */
    dque_qnode     *node;                       /* pointer to deleted node      */

    if (queue == NULL_QUEUE) {                  /* invalid queue pointer?       */
        errcode = DQUEERR_NOQUEUE;
    } else if (data == (void **)NULL) {         /* invalid data pointer?        */
        errcode = DQUEERR_NODATAP;
    } else if (gethead(queue) == NULL_NODE){/* empty queue?                 */
        *data    = (void *)NULL;                /* technically OK, but no data  */
    } else if ((errcode = dque_mydelete( &gethead( queue ), &node, NOROTATE,
&gethcnt( queue )) != DQUEERR_NOERR) {
        ;                                       /* NOROTATE to delete from front*/
    } else if ((errcode = dque_myinsert( &getfree( queue ), node, NOROTATE,
&getfcnt( queue )) == DQUEERR_NOERR) {
    spin_lock(&sl);                 /* Spinlock-Lock or Atomic or compare & swap*/
        *data    = getdata( node );
     spin_unlock(&sl);              /* Spinlock-unLock or Atomic or compare & swap*/
        setdata( node, (void *)NULL );      /* set to null for safety       */
    }

    return (errcode);
}
```

3. *References:*

1. Lecture slides, ECE/CS5510 Multiprocessor programming (Fall'2023).
2. Chapter 7, The Art of Multiprocessor Programming, Revised Reprint by Maurice Herlihy and Nir Shavit.
3. Algorithms for scalable synchronization on shared-memory multiprocessors, ACM Transactions on Computing systems (1991) by JOHN M. MELLOR-CRUMMEY and MICHAEL L. SCOTT.
4. A protocol for wait-free, atomic, multi-reader shared variables. R. N. Wolfe. In PODC '87: Proc. of the Sixth Annual ACM Symposium on Principles of Distributed Computing, pp. 232–248, NY, USA, 1987, ACM Press
5. T. Craig. Building FIFO and priority-queueing spin locks from atomic swap. Technical Report TR 93-02-02, University of Washington, Department of Computer Science, February 1993.
6. CLH Mutex - An alternative to the MCS Lock, Concurrency Freaks:
7. Double Ended Queue: https://en.wikipedia.org/wiki/Double-ended_queue
8. Programming in Cilk : https://cilk.mit.edu/programming/
9. Maged Micheal, "CAS-Based Lock-Free Algorithm for Shared Deques"
10. Robert D. Blumofe et.al, "Cilk: An Efficient Multithreaded Runtime System"
11. Cilk Tests, https://github.com/OpenCilk/applications
12. https://concurrencyfreaks.blogspot.com/2014/05/exchg-mutex-alternative-to-mcs-lock.html
13. GCC Atomic Builtins : https://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/Atomic-Builtins.html