**Assignment #2: Queues without Frontiers**

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

**Link to github :**
**https://github.com/Virginia-Tech-CS-Systems-Courses/queues-without-frontiers-babbons_from_the_bag_end**
**Mirror Repository:**
**https://github.com/lalitprasadperi/Queues_without_frontiers_group15**

# Performance - Sanity Tests

- **Test #1:** One Million Enqueues with enqueuers threads scaling - {1,2,4,8, 16}
- **Test #2:** One Million Dequeues with dequeuers threads scaling - {1,2,4,8,16}
- **Test #3**: One million Enqueus followed by barrier() followed by One million Enqueues and One Million Dequeus for threads scaling - {1 to 16}
- **Scan Test** : Enabled at the end of each test with a single thread parsing the queue contents and matching with expected values.
-

**Runtime reported in milliseconds.**

```
--------------------------------------
test test-spinlock-ttas
--------------------------------------
0.008443        0.008443        0.005882
0.015395        0.013996        0.013492
0.013309        0.013767        0.015805
0.015732        0.014563        0.014665
0.014191        0.014186        0.013758
--------------------------------------
TEST#1 ENQ-TTAS // 1 Million Enqueues
--------------------------------------
0.026050        0.028310        0.027295
0.182433        0.210491        0.194317
0.248955        0.242115        0.238337
0.238571        0.252537        0.248661
0.282196        0.292524        0.284150
--------------------------------------
TEST#2 DEQ-TTAS // 1 Million Dequeues
--------------------------------------
0.047583        0.045093        0.047033
0.173810        0.178550        0.218887
0.255360        0.233399        0.194696
0.215073        0.191348        0.203890
0.202601        0.206122        0.212884
--------------------------------------
TEST#3 ENQ.ENQ.DEQ-TTAS // 1 Million Enqueues + 1 Million Enques-> 1 Million Dequeues
--------------------------------------
0.101417        0.085867        0.085486
0.588876        0.596225        0.568995
0.659896        0.842141        0.608468
0.814220        0.814813        0.808578
0.921437        0.850573        0.903723
```

```
--------------------------------------
TEST#1 ENQ-WAIT-FREE // 1 Million Enqueues
--------------------------------------
0.000198        0.000139        0.000168
0.000205        0.000249        0.000206
0.000360        0.000401        0.000405
0.000687        0.000643        0.000683
0.001247        0.001422        0.001160
--------------------------------------
TEST#2 DEQ-WAIT-FREE // 1 Million Dequeues
--------------------------------------
0.000178        0.000139        0.000185
0.000254        0.000191        0.000256
0.000386        0.000393        0.000385
0.000685        0.000652        0.000655
0.001182        0.001237        0.001227
--------------------------------------
TEST#3 ENQ.ENQ.DEQ-WAIT-FREE // 1 Million Enqueues + 1 Million Enques-> 1 Million Dequeues
--------------------------------------
0.000286        0.000415        0.000279
0.000588        0.000561        0.000515
0.000859        0.001008        0.000925
0.002248        0.002269        0.002260
0.004374        0.004236        0.004405
--------------------------------------
ENQ.ENQ.DEQ-LOCK-FREE
--------------------------------------
==========================================
  Benchmark: ./msqueue
  Number of processors: 2
  Number of operations: 10000000
  #1 elapsed time: 286.59 ms
  #2 elapsed time: 291.09 ms
  #3 elapsed time: 290.70 ms
  #4 elapsed time: 296.75 ms
  #5 elapsed time: 282.13 ms
  Steady-state iterations: 1~5
  Coefficient of variation: 0.02
  Number of measurements: 5
  Mean of elapsed time: 289.45 ms
==========================================
  Benchmark: ./msqueue
  Number of processors: 4
  Number of operations: 10000000
  #1 elapsed time: 2648.38 ms
  #2 elapsed time: 2654.93 ms
  #3 elapsed time: 2670.64 ms
  #4 elapsed time: 2656.92 ms
  #5 elapsed time: 2636.36 ms
  Steady-state iterations: 1~5
  Coefficient of variation: 0.00
  Number of measurements: 5
  Mean of elapsed time: 2653.45 ms
==========================================
  Benchmark: ./msqueue
  Number of processors: 8
  Number of operations: 10000000
  #1 elapsed time: 16356.81 ms
  #2 elapsed time: 17073.14 ms
  #3 elapsed time: 16992.10 ms
  #4 elapsed time: 16970.33 ms
  #5 elapsed time: 17258.28 ms
```

```
  Steady-state iterations: 1~5
  Coefficient of variation: 0.02
  Number of measurements: 5
  Mean of elapsed time: 16930.13 ms
=========================================
=========================================
  Benchmark: ./msqueue
  Number of processors: 8
  Number of operations: 10000000
  #1 elapsed time: 16356.81 ms
  #2 elapsed time: 17073.14 ms
  #3 elapsed time: 16992.10 ms
  #4 elapsed time: 16970.33 ms
  #5 elapsed time: 17258.28 ms
  Steady-state iterations: 1~5
  Coefficient of variation: 0.02
  Number of measurements: 5
  Mean of elapsed time: 16930.13 ms
=========================================
```

## Locked based Queues -TTAS

This queue implements a linked list based queue with a pointer to the next object updated with each enqueue and dequeue. This implementation uses two stage locks (coarse grained and fine grain) - TTAS locks and mutex_locks.

## Enqueue:

```
bool queue_enq(queue_t *queue, void const *src)
{
    bool success;
    void *dest;
    if ((queue == NULL) || (queue_full(queue)) || (src == NULL))
    {
        success = false;
    }
    else
    {
        spin_lock(&sl);  // TTAS LOCK
        dest = index_to_address(queue, queue->in);
        memcpy(dest, src, queue->sizeof_element);
        queue->in = (queue->in + 1) % queue->capacity;
        success = true;
        spin_unlock(&sl); // TTAS UNLOCK
    }
    return success;
}
```

## Dequeue:

```
bool queue_deq(queue_t *queue, void *dest)
{
    bool success;

    if ((queue == NULL) || (queue_empty(queue)))
    {
        success = false;
    }
    else
    {
       spin_lock(&sl); // TTAS lock
        if (dest != NULL)
        {
            queue_front_internal(queue, dest, false);
        }
        queue->out = (queue->out + 1) % queue->capacity;
        success = true;
       spin_unlock(&sl); // TTAS unlock
    }

}
```

# Lock-Free Concurrent Queue Implementation:

Our implementation is based upon the Michael-Scott Queue[1]. The Michael-Scott algorithm implements a queue as a singly-linked list with head and tail pointers, head is always pointing to a dummy node. We decided to use hazard pointers[2] instead of the counter solution in the original implementation. Our implementation of hazard pointers is taken from [3]. This allows us to use an atomic compare and swap operation without running into the ABA problem.

**Enqueue:**

The enqueue method will first allocate a new node to contain the data to be enqueued. Enqueue loops until the operation is successfully completed. For each iteration of the loop, we check that tail and next are consistent, then check if tail is pointing to the last node in the list. If the tail is pointing to the last node in the list, we attempt to insert the new node at the end of the list, otherwise we will move the tail to the next node. After the loop completes, we will try to move the tail to the inserted node which should be at the end of the list.

```
void enqueue(queue_t * q, handle_t * handle, void * data)
{
 node_t * node = malloc(sizeof(node_t));
 node->data = data;
 node->next = NULL;
 node_t * tail;
 node_t * next;
 while (1) {
  tail = hzdptr_setv(&q->tail, &handle->hzd, 0);
  next = tail->next;
  if (tail == q->tail) {
   if (next == NULL) {
    if (CAS(&tail->next, &next, node)) break;
   } else {
    CAS(&q->tail, &tail, next);
   }
  }
 }
 CAS(&q->tail, &tail, node);
}
```

**Dequeue:**

The dequeue method runs a loop that iterates until the operation is successfully completed. First, we check that head, tail, and next are consistent then check if the queue is empty or if tail is not pointing to the last element in the list. If the queue is empty, we return false. Otherwise, we try to advance the tail to point to the last element in the list. If the queue is not empty and the tail is in the correct position, we move head to point to the next node in the list and exit the loop.

```
bool dequeue(queue_t * q, handle_t * handle)
{
 node_t * head;
 node_t * tail;
 node_t * next;
 while (1) {
  head = hzdptr_setv(&q->head, &handle->hzd, 0);
  tail = q->tail;
  next = hzdptr_set(&head->next, &handle->hzd, 1);
```

```
  if (head == q->head) {
    if (head == tail) {
      if (next == NULL) {
        return false;
      }
      CAS(&q->tail, &tail, next);
    } else {
      if (CAS(&q->head, &head, next)) break;
    }
  }
 }
 hzdptr_retire(&handle->hzd, head);
 return true;
}
```

## Wait-Free Queue

Wait free queue uses a linked list based multi-enqueur multi-dequeuer algorithm which is inspired by MC-Queue, wCQ and producer-consumer queues but uses a different approach by using simpler atomic operations like compare_and_set or compare_and_add.

### Enqueue:

```
bool wfq_enqueue(struct wfq *q, void *obj)
{
        size_t count = atomic_fetch_add_explicit(&q->count, 1,
memory_order_acquire);
        if(count >= q->max) {
                /* back off, queue is full */
                atomic_fetch_sub_explicit(&q->count, 1, memory_order_release);
                return false;
        }
        /* increment the head, which gives us 'exclusive' access to that
element */
        size_t head = atomic_fetch_add_explicit(&q->head, 1,
memory_order_acquire);
        assert(q->buffer[head % q->max] == 0);
        void *rv = atomic_exchange_explicit(&q->buffer[head % q->max], obj,
memory_order_release);
        assert(rv == NULL);
        return true;
}
```

### Dequeue:

```
void *wfq_dequeue(struct wfq *q)
{
        void *ret = atomic_exchange_explicit(&q->buffer[q->tail], NULL,
memory_order_acquire);
        if(!ret) {
                //a thread is adding to the queue, but hasn't done the
atomic_exchange yet
                // to actually put the item in. Act as if nothing is in the
queue.
                return NULL;
        }
        if(++q->tail >= q->max)
                q->tail = 0;
        size_t r = atomic_fetch_sub_explicit(&q->count, 1,
memory_order_release);
        assert(r > 0);
        return ret;
}
```

**Known Limitations: Scan-test assertion/failures with higher thread counts (>8) with Test #3. Though this behavior is only observed intermittently when server load is higher.**

## Bibliography:

[1]: Michael, Maged M., and Michael L. Scott. "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms." *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*. 1996.
[2]: Michael, Maged M. "Hazard pointers: Safe memory reclamation for lock-free objects." IEEE Transactions on Parallel and Distributed Systems 15.6 (2004): 491-504.
[3]: https://github.com/chaoran/fast-wait-free-queue/tree/master
[4] Maged M. Michael, “High Performance Dynamic Lock-Free Hash Tables and List-Based Sets”
[5] John D. Valois, “Lock-Free Linked Lists Using Compare-and-Swap”
[6] Ruslan Nikolaev., and Binoy Ravindran. “wCQ: A Fast Wait-Free Queue with Bounded Memory Usage”
[7] Chaoran Yang John., Mellor-Crummey, “A Wait-free Queue as Fast as Fetch-and-Add”
[8] Alex Kogan and Erez Petrank, “Wait-Free Queues With Multiple Enqueuers and Dequeuers”
[9] Lock-Free queues, Concurrency Freaks:
https://concurrencyfreaks.blogspot.com/2014/05/exchg-mutex-alternative-to-mcs-lock.html
[10] GCC Atomic Builtins : https://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/Atomic-Builtins.html