Indian Institute of Technology Hyderabad
Computer Science & Engineering Department
CS5300 Parallel & Concurrent Programming

Different Implementations of Lock Free FIFO Queue

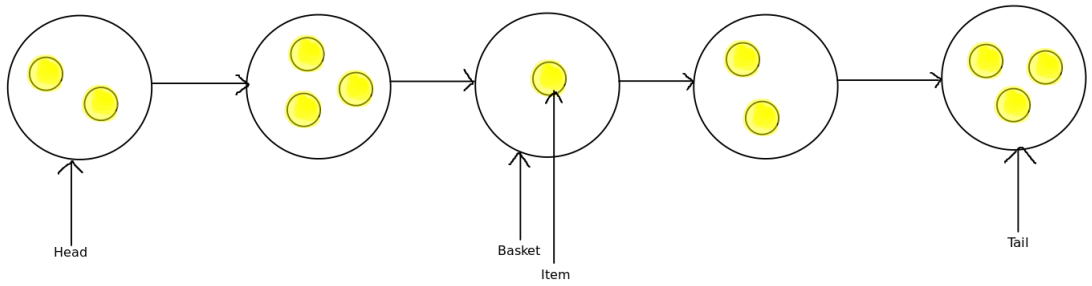
Ajinkya Bokade (CS17BTECH11001)
& Atharva Sarage (CS17BTECH11005)

December 23, 2020

Baskets Queue

Design of Queue

Basket Queue is a lock free linearizable highly concurrent linearizable FIFO queue. It maintains baskets of mixed-order items instead of standard totally ordered list. It parallelizes enqueue operations among different baskets by allowing different enqueue operations in different baskets to execute parallely. Nodes' order in baskets isn't specified while enqueuing .
Linearizability allows us to reorder concurrent operations. This motivates the idea of making basket queue linearizable. Overlapping enqueue operations can be enqueued into queue as a group (basket) without specifying order of the nodes being enqueued. Thus, nodes of same basket can be dequeued in any order. Thus the order of nodes dequeued from a basket can be assigned as the order in which the nodes were enqueued in that basket.
It allows parallelism among different baskets by allowing threads to insert nodes in different baskets simultaneously.



Basket queue's principles

- Time during which all nodes' enqueue operations overlap is time interval of that basket. This time interval is nothing but time when first enqueue operation among a group of concurrent enqueue operations is successfully completed. (Among threads executing CAS concurrently, first thread (winner) to perform CAS successfully)
- Order of baskets is decided by their respective time interval.
- Dequeue operations of nodes of a basket happen after the time interval where all enqueue operations of that basket have been overlapped.
- Dequeue operations of nodes of previous basket precede dequeue operations of nodes of next basket

Basket queue's properties defining FIFO order

- There is no need to specify nodes' order of a basket. (Order of nodes in which concurrent enqueue operations happen for a basket is assigned to be same as order of nodes in which dequeue happens.)
- FIFO order is the nodes' order in different baskets.

Queue Initialization

```
struct pointer_t {
  <ptr, deleted, tag>: <node_t *, boolean, unsigned integer>
};

struct node_t {                              struct queue_t {
  data_type value;                             pointer_t tail
  pointer_t next;                              pointer_t head
};                                           };

void init_queue(queue_t* q)
I01: node_t* nd = new_node()       # Allocate a new node
I02: nd->next = <null, 0, 0>       # next points to null with tag 0
I03: q->tail = <nd, 0, 0>;         # tail points to nd with tag 0
I04: q->head = <nd, 0, 0>;         # head points to nd with tag 0
```

Fig. 4. Types, structures and initialization

Basket queue is implemented as an explicit linked list of nodes with head and tail pointers. Queue's head always points to dummy nodes, which might be followed by sequence of marked (logically deleted) nodes. Queue's tail points either to a node is last basket or second last basket.
Two structures are defined named pointer_t and node_t.
pointer_t has member fields ptr which stores the node_t pointer, boolean deleted which denotes whether node is logically deleted or not, unsigned integer tag which is used to solve ABA problem.
node_t has member fields value which stores the value stored in the node, atomic pointer_t next which stores the pointer of its next node.
Queue's head and tail are declared as atomic pointer_t and initialized both pointing to dummy node with deleted field 0 and tag field 0.

Enqueue algorithm

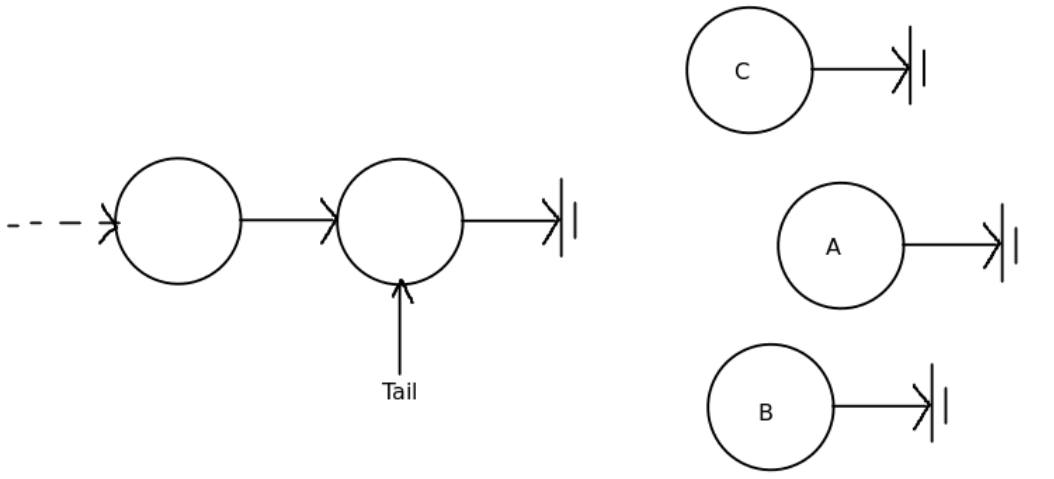
```
E01: nd = new_node()
E02: nd->value = val
E03: repeat:
E04:    tail = Q->tail
E05:    next = tail.ptr->next
E06:    if (tail == Q->tail)):
E07:       if (next.ptr == NULL):
E08:          nd->next = <NULL, 0, tail.tag+2>
E09:          if CAS(&tail.ptr->next, next, <nd, 0, tail.tag+1>):
E10:             CAS(&Q->tail, tail, <nd, 0, tail.tag+1>
E11:             return True
E12:          next = tail.ptr->next
E13:          while((next.tag==tail.tag+1) and (not next.deleted)):
E14:             backoff_scheme()
E15:             nd->next = next
E16:             if CAS(&tail.ptr->next, next, <nd, 0, tail.tag+1>):
E17:                return True
E18:             next = tail.ptr->next;
E19:       else:
E20:          while ((next.ptr->next.ptr != NULL) and (Q->tail==tail)):
E21:             next = next.ptr->next;
E22:          CAS(&Q->tail, tail, <next.ptr, 0, tail.tag+1>)
```

Fig. 5. The enqueue operation

(E01-02) It is initialization of new node with given value. (E04-05) First, enqueuer thread obtains tail which is thread local from queue's tail pointer. next which is also thread local denotes the pointer of tail's next node. (E06) It checks if tail pointer and queue's tail pointer are same or not. If they are not same, it means in the time interval thread assigns local tail as queue's tail and then checks if they are same or not, either other threads have enqueued successfully thereby changing queue's tail or other threads have changed queue's tail pointer to point to closest unmarked node. (E07) It checks if tail's next pointer is NULL or not. If it is NULL, it means queue's tail is not lagging and it is pointing to last node in the queue. (E08) If tail points to last node, then the thread assigns next pointer of new node to point to a NULL node with deleted field 0 and tag field as 2+tail field. (E09-E11) All the threads, who try the CAS operation will fall in same bucket, since all these threads are concurrently enqueueing to the queue. Thus, the thread whose CAS operation of comparing atomically local tail's next to next and assigning next pointer of local tail to its new node succeeds will be winner. That is, this thread will append its node to queue's tail. (E10-E11) Winner thread does CAS on queue's tail to atomically check if its equal to local tail and assigns queue's tail to point to its new node and returns true, since it successfully appended its new node to queue. (E13-E18) Other threads who failed CAS operation on E09 further checks the nodes in queue by skipping logically deleted node and does CAS on first unmarked node (E16-E17) and if it succeeds, then it appends its new node before the local tail it has read initially. Thus all the nodes which failed CAS operation on E09 append their nodes in LIFO manner before the node inserted by winner thread of CAS operation (see below figures). (E19-E22) If queue's tail is lagging i.e it is not pointing to last node, then last node is searched and queue tail is fixed. Thus fixing queue's tail, thread retries its enqueue operation.

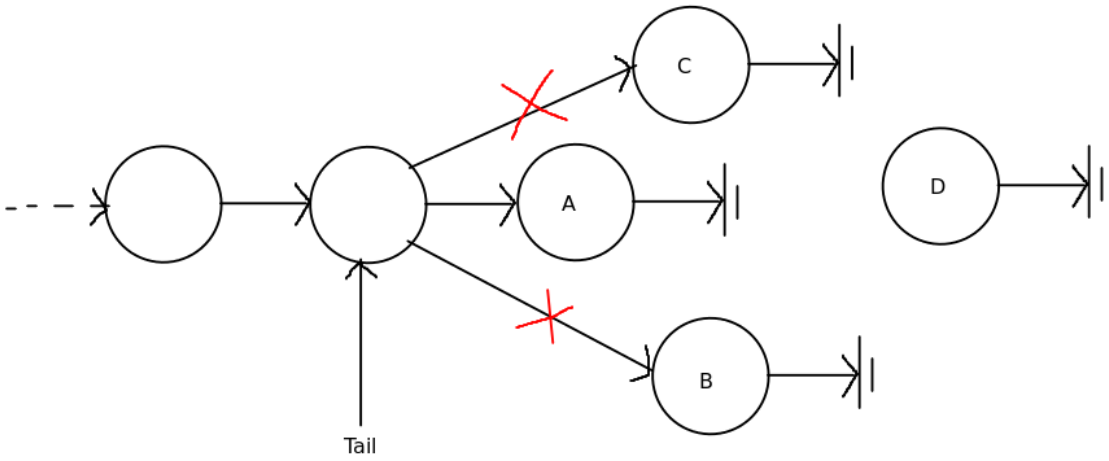
Method is lock-free since threads overlapping in time interval of winner thread of CAS operation are only finite. Thus, they will eventually complete their insertion (E13-E18). Threads which observed that their next node is not NULL (E19-E22) fixes the tail pointer of queue and then retry their process of enqueue.

Enqueue example

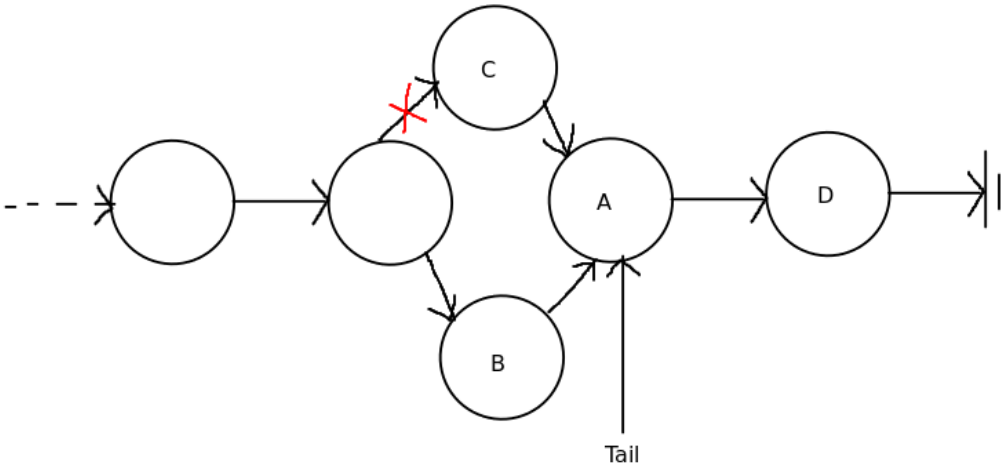


Concurrent enqueuers thread
A, B, C

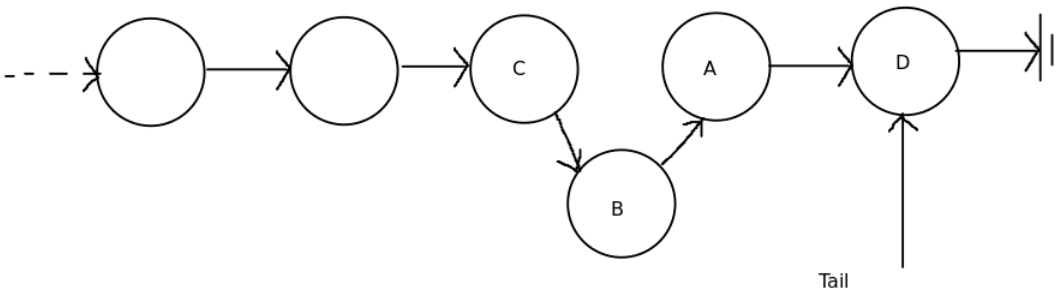
Each thread A, B, C checks if next field of tail node is NULL and tries to change it atomically to point to new node's address.



Thread A succeeds in enqueuing its node. Threads B and C fails the CAS operation. Thus thread B and C fall on same time interval and will thus enqueue their nodes in same basket created by winner thread A, thus will retry to enqueue. Another enqueuer thread D has come but since it comes after A's enqueue, it will enqueue its node in next basket.



Thread B now succeeds in CAS operation and thus it enqueues its node after the head it has read initially. Thus, it is inserted between node inserted by thread A and old head read by thread B. Thread C again fails CAS operation, thus retries enqueueing. Thread D enqueues its node in next basket i.e. after node inserted by thread A.



Thread C finished its enqueue operation and inserts node between head it has read initially and node inserted by thread B.

Dequeue algorithm

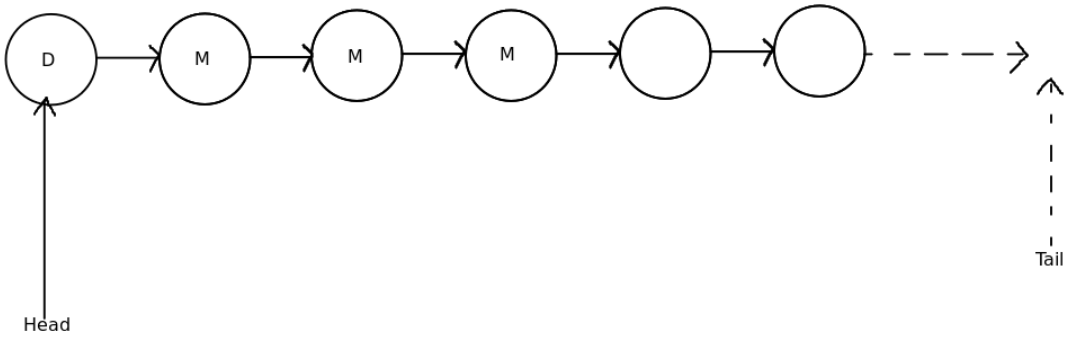
```
const MAX_HOPS = 3 # constant

data_type dequeue(queue_t* Q)

D01: repeat
D02:    head = Q->headf
D03:    tail = Q->tail
D04:    next = head.ptr->next
D05:    if (head == Q->head):
D06:       if (head.ptr == tail.ptr)
D07:          if (next.ptr == NULL):
D08:             return 'empty'
D09:          while ((next.ptr->next.ptr != NULL) and (Q->tail==tail)):
D10:             next = next.ptr->next;
D11:          CAS(&Q->tail, tail, <next.ptr, 0, tail.tag+1)
D12:       else:
D13:          iter = head
D14:          hops = 0
D15:          while ((next.deleted and iter.ptr != tail.ptr) and (Q->head==head)):
D16:             iter = next
D17:             next = iter.ptr->next
D18:             hops++
D19:          if (Q->head != head):
D20:             continue;
D21:          elif (iter.ptr == tail.ptr):
D22:             free_chain(Q, head, iter)
D23:          else:
D24:             value = next.ptr->value
D25:             if CAS(&iter.ptr->next, next, <next.ptr, 1, next.tag+1>):
D26:                if (hops >= MAX_HOPS):
D27:                   free_chain(Q, head, next)
D28:                return value
D29:             backoff-scheme()
```

Fig. 6. The dequeue operation

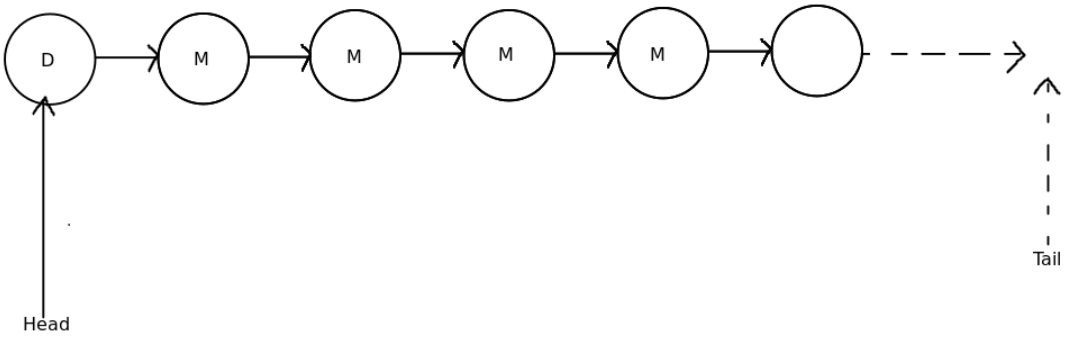
MAX_HOPS is a constant declared as 3 which denotes the maximum number of logically deleted nodes in queue after which the logically deleted nodes will be physically deleted.
(D02-D04) Thread assigns local pointers head and tail as queue's head and tail respectively. It also assigns local pointer next as next pointer of its local head. (E05) It checks if head pointer and queue's head pointer are same or not. If they are not same, it means other threads have already deleted nodes and changed queue's head. Thus it retries. (E06-E11) If head and tail pointers are same that is they point to dummy node, then either (D07-D08) it means queue is empty, thereby returning empty or (D09-D11) queue's tail is lagging i.e. not pointing to last node of queue, thereby changing it to last node of queue and retries. (D12) Queue is not empty. (D13-D14) It assigns iter pointer pointing to local head pointer and initializes hop to 0. (D15-D18) It hops to next node until it finds first non logically-deleted node. (D19-D20) If it observes that at this point head is not same as queue's head, it means other threads have deleted nodes and changed queue's head pointer. Thus it retries. (D21-D22) If it finds that tail and iter are same i.e. it has reached end of list, it calls free_chain which physically deletes logically deleted nodes between head and iter. (D23-D29) It retrieves value of next pointer. Note next pointer points to non marked node and iter points to previous node of next. It then checks if iter pointer's next is still next or not. If not, it means other threads have changed member fields of next node. If thread succeeds in performing CAS operation, then it marks the node by setting deleted field to true, thereby performing logical deletion. If it observes that more than MAS_HOPS nodes are logically deleted (marked), then it physically deletes those nodes thus reclaiming memory by calling free_chain.
Method is lock free since threads observing head and tail pointing to same node (D06-D11), either returns empty if queue is empty or fixes the tail node of queue. Else, all threads skip over all logically deleted threads and then do CAS (D25) to remove node pointed by next. All other threads who failed CAS (D25) retry their dequeue process.

Dequeue example

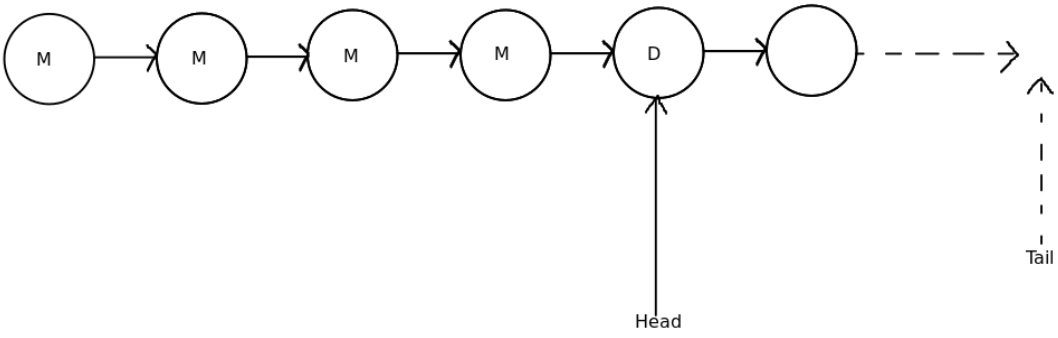
M - marked node (logically deleted), D - dummy node, F - freed node (physical deleted)



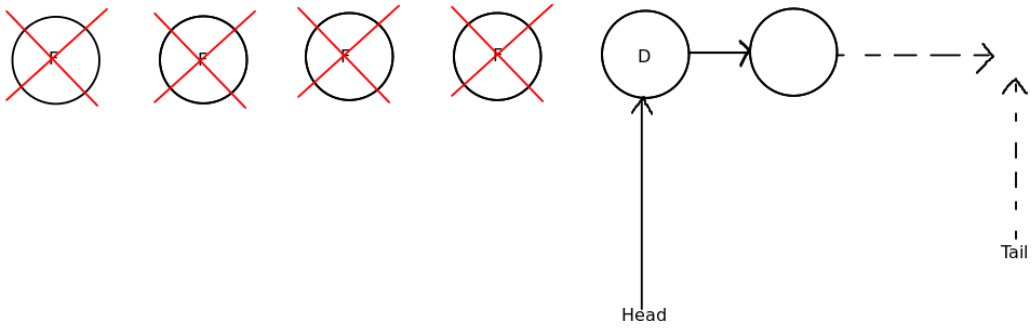
Three Nodes logically deleted (logical deletion)



First non-deleted node is deleted (logical deletion)



Head pointer is advanced to first non-marked node



Physical deletion i.e. Memory is reclaimed

Linearization points and Linearizablity proof

Linearization point of enqueue of nodes inside a particular basket are set inside basket's shared time interval in the order of their respective dequeues i.e. linearization point of enqueues is decided only once items inside the baskets are dequeued.
Linearization point of dequeue returning a value (queue is not empty) is line D23 where next pointer points to non marked node (not logically deleted), thereafter thread will mark it as logically deleted.
Linearization point of dequeue returning empty (queue is empty) is line D04, where it reads the next pointer of head which is NULL.

Claim 1: Enqueue operations of same basket overlap in time
A new basket is created by the winner thread of CAS operation in (E09) of enqueue. Other threads who failed the CAS operation insert nodes in same basket created by winner thread. Thus time interval in which this overlapping enqueue operations happen starts when next pointer of tail node was null and ends when winner threads successfully inserts its node to queue. Winner thread of CAS operation also overlap the same interval too. Thus all enqueue operations of same basket overlap in time.

Claim 2: Baskets are ordered according to order of their respective time intervals
A new basket is created when thread successfully executes CAS operation (E09) and appends nodes to last node of queue. Enqueue operations that failed the CAS operation (E09), retry their enqueue process to insert their nodes at same list position they have read initially (in same basket). Thus, first node of new basket is next to last node of previous basket and last node of basket is being inserted by winner thread of CAS operation.

Claim 3: Linearization points of the dequeue operations of a basket come after the basket's shared time interval when all enqueue operations of same basket overlap
Node must be present in the queue for its dequeue to happen. Basket's first node is inserted into the queue only after the winner thread has succeeded in CAS operation (E09) on tail. Completion of the CAS operation marks the end of shared time interval of overlapping enqueue operations of that basket. Thus nodes can be marked (by dequeuers) only after the basket's time interval.

Claim 4: Nodes of a basket are dequeued before nodes of later baskets i.e. nodes of previous basket are dequeued before nodes of next basket
Nodes are dequeued according to their sequential order in the queue (which in this case, is logically divided into baskets). Since, once the nodes of basket are dequeued i.e. logically deleted, then no more nodes are allowed to be enqueued in the same basket.

Theorem: FIFO queue is linearizable to a sequential FIFO queue
By Claim 2 and 4 above, we know the order in which basket's nodes are dequeued is same as order in which the nodes of these baskets are enqueued. By Claim 3, enqueue operation of a node preceed its dequeue operation. If we linearize the order of enqueue operations of a basket in the order of the nodes dequeued from that basket, then we can linearize the operations and queue becomes linearizable.

Circular Queue

Design of Circular Queue Class

We introduce a new approach of building circular queues using indirection and two queues.

Indirection

The data entries are not stored in the queue itself, insted a queue entry records an index in the array of data. For simplicity we consider the data to be integers.

Unallocated Entries Queue (fq)

This queue is denoted by fq it keeps the indices to unallocated entries of the data array.

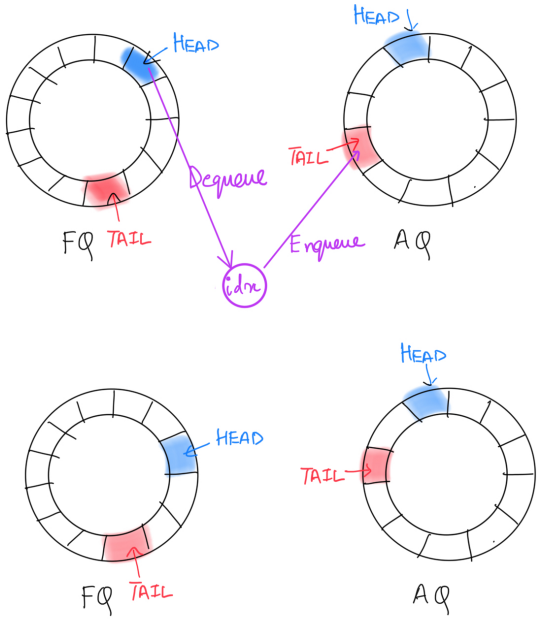
Allocated Entries Queue (aq)

This queue keeps allocated indices which are to be consumed.

Producer

A producer thread will dequeue an index from fq . If NULL / -1 is returned the queue is full o/w will return a valid index in which case we write data to corresponding index and insert the index into aq.

```
// data:  an array of pointers
// aq is initialized empty
// fq is initialized full
1 bool enqueue_ptr(void * ptr)
2     int index = fq.dequeue();
3     if ( index = ∅ ) return False;       // Full
4     data[index] = ptr;
5     aq.enqueue(index);
6     return True;                         // Success
```

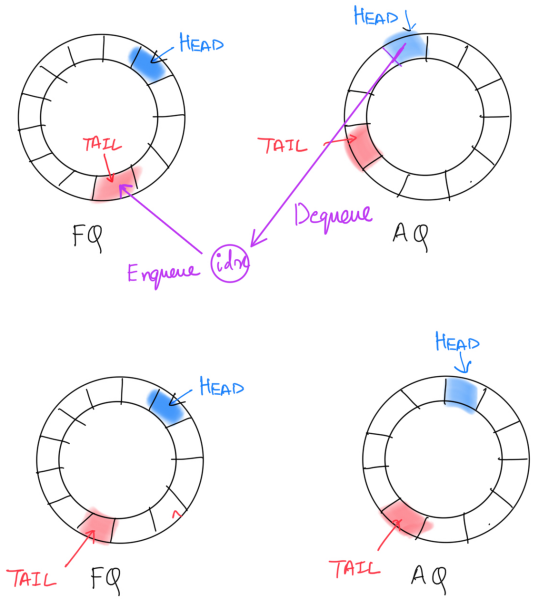


PRODUCER THREAD

Consumer

A consumer thread will dequeue index from aq . If NULL/-1 is returned the queue is empty o/w we get a valid index in which case we read the data from data array and insert the entry back into fq.

```
7  void * dequeue_ptr()
8      int index = aq.dequeue();
9      if ( index = ∅ ) return nullptr;     // Empty
10     ptr = data[index];
11     fq.enqueue(index);
12     return ptr;                          // Success
```



Consumer Thread

Queue Class

- Both the queues have Head and Tail pointers and tail is incremented when new entries are enqueued, Head is incremented when new entry is dequeued.
- An entries array which will store the index and cycle of each entry in the queue.
- The total increments can be represented as i $*$ n $+$ j where n is the capacity of queue. Here i is called the cycle number and j is called index which denotes position in circular array.
- Each queue has an n sized array called entries which will store the cycle and index of the elements in queue
- aq is initialized to be an empty queue. All entries are initialized to cycle 0 and Head = n , Tail = n
- fq is initialized to be a full queue. It will contains all the n entries. (We initialize it as empty queue and enqueue all the n indices)
- It has 2 methods enqueue() and dequeue(). Lets see them !!

Enqueue Algorithm

```
2  forall entry_t Ent ∈ Entries[n] do
3      Ent = { .Cycle: 0, .Index: 0 };
4  void enqueue(int index)
5      do
6          T = Load(&Tail);
7          j = Cache_Remap(T mod n);
8          Ent = Load(&Entries[j]);
9          if ( Cycle(Ent) = Cycle(T) )
10             CAS(&Tail, T, T + 1);    // Help to
11             goto 6;                  // move tail
12         if ( Cycle(Ent) + 1 ≠ Cycle(T) )
13             goto 6;        // T is already stale
14         New = { Cycle(T), index };
15     while !CAS(&Entries[j], Ent, New);
16     CAS(&Tail, T, T+1);     // Try to move tail
```

- Note that **enqueue does not need to check if a queue is full**. It is called only when an available entry out of n exists. (Only if entry can be dequeued from fq)
- New entry (Tail / N , value) is enqueued at index = (tail % N)
- **Claim** Tail must be one cycle in front of the current entry.
  Proof Tail is initialized to n and entry's cycle are initialized to 0. So the property is satisfied at the start (first round of enqueues).
  Now let's look at second round of enqueues. We know that After every successful enqueue operation tail is incremented by 1. So when we again come back at this point (2nd round) tail would have been incremented n times
  $\implies$ it would be 1 cycle ahead of the entry.
- If Tail's cycle number is same as entry's cycle number $\implies$ Some other thread has enqueued but has not advanced tail. So we increase tail for global progress.
- If both the conditions are satisfied we are reading an old value of tail (n enqueues happened). So we again fetch the new value of tail.
- If the conditions are satisfied we do a Compare & Swap operation to update the entry at this index with new entry (Cycle(Tail) , value)

Dequeue Algorithm

```
17 int dequeue()
18     do
19         H = Load(&Head);
20         j = Cache_Remap(H mod n);
21         Ent = Load(&Entries[j]);
22         if ( Cycle(Ent) ≠ Cycle(H) )
23             if ( Cycle(Ent) + 1 = Cycle(H) )
24                 return ∅;        // Empty queue
25             goto 19;        // H is already stale
26     while !CAS(&Head, H, H+1);
27     return Index(Ent);
```

- Dequeue is possible only if queue is not empty
- **Claim 1** Cycle number of current entry and Head should match.
  Claim 2: The values of H^b and T^b before an item b is enqueued are same. Using this above claim also holds
  Proof: We Proove by Induction on no of enqueues
  Base Case Initially both Tail and Head = N.
  Inductive Step Let T_a and H_a be the value of Head and Tail before a is enqueued.
  After a is enqueued. cycle(a) = cycle(T_a) and $T'_a = T_a + 1$
  After a is dequeued cycle(a) = cycle(H_a) ($\because T_a = H_a$)
  and $H'_a = H_a + 1 \implies H'_a = T'_a$
  This also proved the above claim $\because$ cycle(a) = cycle(H_a) ($\because T_a = H_a$)
  Hence Proved
- If Head is one cycle ahead $\implies$ empty queue in this case we return NULL
- In any other case we are reading an old value. So we again fetch the new value of Head

The Algorithm is Lock-Free

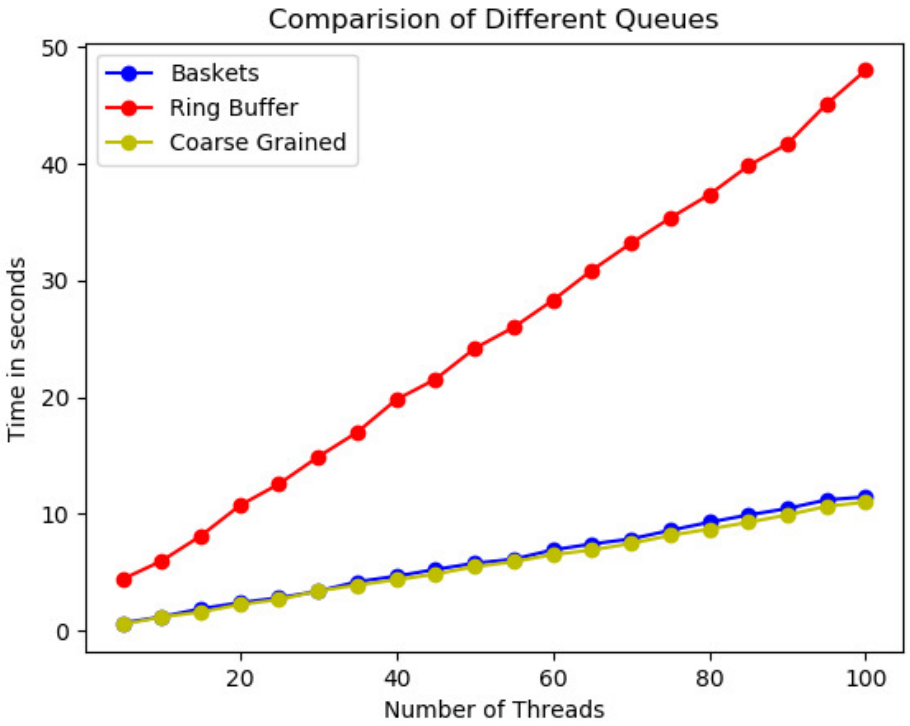
Proof The Circular Queue algorithm has 2 unbounded loops. First one in enqueue() method and other in the dequeue method. **Case 1**

- If CAS operation fails in the enqueue it again repeats the loops, $\implies$ the entry Entries[j] was updated by another thread who called enqueue method $\because$ dequeue method didn't modify entries array.
- $\implies$ That other thread was making progress and it succeded in its enqueue operation.

Case 2

- If CAS operation fails in the dequeue it again repeats the loops, $\implies$ the Head pointer was updated by another thread who called dequeue method $\because$ enqueue method didn't modify head pointer.
- $\implies$ That other thread was making progress and it succeded in its dequeue operation

Results & Graphs



We used the 50% enqueues and 50% dequeues for performing our experiments.
We measured the total time taken to execute 10^5 requests either enqueue , dequeue per thread where threads varied from 5 to 1000.
Execution Machine Specifications

- Intel Core i5 7th Gen 7200U 4 cores
- Base Clock Speed 2.5 GHz
- Burst Clock Speed 3.1 GHz Cache 3MB
- 16GB RAM

We observed that **Coarse Grained Queue performed almost the same as Baskets Queues both of which performed better than Circular Queue.**

It was observed the Circular Queue Algorithm although didn't use locks but was slow. The reason might be the algorithm did not use full 4 cores simultaneously, CPU time was only 1.5 x to 2x more than real execution time.
Whereas both for coarse grained queue and baskets queue all the 4 cores were used parallely as CPU time was almost 3.5x to 4.x more than real execution time.

*(For timing calculations printing time was not considered)

References

- The Baskets Queue by Moshe Hoffman1, and Nir Shav (Baskets Queue)
- A Scalable, Portable, and Memory-Efficient Lock-Free FIFO Queue by Ruslan Nikolaev (Circular Queue)