Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms

Pseudocode from article of the above name in *PODC96* (with two typos corrected), by Maged M. Michael and Michael L. Scott. Corrected version also appeared in *JPDC*, 1998.

The non-blocking concurrent queue algorithm performs well on dedicated as well as multiprogrammed multiprocessors with and without contention. The algorithm requires a universal atomic primitive, *CAS* or *LL/SC*. It depends for memory management on a type-preserving allocator that never reuses a queue node as a different type of object, and never returns memory to the operating system. If this is unacceptable in a given context, the code can be modified to incorporate hazard pointers, epoch-based reclamation, or interval-based reclamation.

The two-lock concurrent queue algorithm performs well on dedicated multiprocessors under high contention. Useful for multiprocessors without a universal atomic primitive.

Non-Blocking Concurrent Queue Algorithm

```
structure pointer_t {ptr: pointer to node_t, count: unsigned integer}
structure node_t {value: data type, next: pointer_t}
structure queue_t {Head: pointer_t, Tail: pointer_t}

initialize(Q: pointer to queue_t)
   node = new_node()          // Allocate a free node
   node->next.ptr = NULL      // Make it the only node in the linked list
   Q->Head.ptr = Q->Tail.ptr = node   // Both Head and Tail point to it

enqueue(Q: pointer to queue_t, value: data type)
 E1:   node = new_node()      // Allocate a new node from the free list
 E2:   node->value = value    // Copy enqueued value into node
 E3:   node->next.ptr = NULL  // Set next pointer of node to NULL
 E4:   loop                   // Keep trying until Enqueue is done
 E5:      tail = Q->Tail      // Read Tail.ptr and Tail.count together
 E6:      next = tail.ptr->next       // Read next ptr and count fields together
 E7:      if tail == Q->Tail  // Are tail and next consistent?
             // Was Tail pointing to the last node?
 E8:         if next.ptr == NULL
                // Try to link node at the end of the linked list
 E9:            if CAS(&tail.ptr->next, next, <node, next.count+1>)
E10:               break      // Enqueue is done.  Exit loop
E11:            endif
E12:         else              // Tail was not pointing to the last node
                // Try to swing Tail to the next node
E13:            CAS(&Q->Tail, tail, <next.ptr, tail.count+1>)
E14:         endif
E15:      endif
E16:   endloop
       // Enqueue is done.  Try to swing Tail to the inserted node
E17:   CAS(&Q->Tail, tail, <node, tail.count+1>)

dequeue(Q: pointer to queue_t, pvalue: pointer to data type): boolean
```

```
 D1:   loop                         // Keep trying until Dequeue is done
 D2:      head = Q->Head            // Read Head
 D3:      tail = Q->Tail            // Read Tail
 D4:      next = head.ptr->next     // Read Head.ptr->next
 D5:      if head == Q->Head        // Are head, tail, and next consistent?
 D6:         if head.ptr == tail.ptr // Is queue empty or Tail falling behind?
 D7:            if next.ptr == NULL  // Is queue empty?
 D8:               return FALSE      // Queue is empty, couldn't dequeue
 D9:            endif
                // Tail is falling behind.  Try to advance it
D10:            CAS(&Q->Tail, tail, <next.ptr, tail.count+1>)
D11:         else                   // No need to deal with Tail
                // Read value before CAS
                // Otherwise, another dequeue might free the next node
D12:            *pvalue = next.ptr->value
                // Try to swing Head to the next node
D13:            if CAS(&Q->Head, head, <next.ptr, head.count+1>)
D14:               break             // Dequeue is done.  Exit loop
D15:            endif
D16:         endif
D17:      endif
D18:   endloop
D19:   free(head.ptr)              // It is safe now to free the old node
D20:   return TRUE                   // Queue was not empty, dequeue succeeded
```

Two-Lock Concurrent Queue Algorithm

```
structure node_t {value: data type, next: pointer to node_t}
structure queue_t {Head: pointer to node_t, Tail: pointer to node_t,
                   H_lock: lock type, T_lock: lock type}

initialize(Q: pointer to queue_t)
   node = new_node()           // Allocate a free node
   node->next = NULL           // Make it the only node in the linked list
   Q->Head = Q->Tail = node    // Both Head and Tail point to it
   Q->H_lock = Q->T_lock = FREE       // Locks are initially free

enqueue(Q: pointer to queue_t, value: data type)
   node = new_node()           // Allocate a new node from the free list
   node->value = value                // Copy enqueued value into node
   node->next = NULL           // Set next pointer of node to NULL
   lock(&Q->T_lock)            // Acquire T_lock in order to access Tail
      Q->Tail->next = node     // Link node at the end of the linked list
      Q->Tail = node           // Swing Tail to node
   unlock(&Q->T_lock)          // Release T_lock

dequeue(Q: pointer to queue_t, pvalue: pointer to data type): boolean
   lock(&Q->H_lock)            // Acquire H_lock in order to access Head
      node = Q->Head           // Read Head
      new_head = node->next    // Read next pointer
      if new_head == NULL      // Is queue empty?
         unlock(&Q->H_lock)    // Release H_lock before return
         return FALSE          // Queue was empty
      endif
      *pvalue = new_head->value       // Queue not empty.  Read value before release
      Q->Head = new_head       // Swing Head to next node
   unlock(&Q->H_lock)          // Release H_lock
   free(node)                  // Free node
   return} TRUE                // Queue was not empty, dequeue succeeded
```

Last Change: 30 January 2008 / scott@cs.rochester.edu