CS510 Concurrent Systems

Jonathan Walpole

Portland State University

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

Outline

- Background
- Non-Blocking Queue Algorithm
- Two Lock Concurrent Queue Algorithm
- Performance
- Conclusion

Background

- Locking vs Lock-free
- Blocking vs non-blocking
- Delay sensitivity
- Dead-lock
- Starvation
- Priority inversion
- Live-lock

Linearizability

- A concurrent data structure gives an external observer the illusion that its operations *take effect instantaneously*
- There must be a specific point during each operation at which it *takes effect*
- All operations on the data structure can be ordered such that *every viewer agrees on the order*
- Linearizability is about equivalence to sequential execution, it is not a condition of correctness!

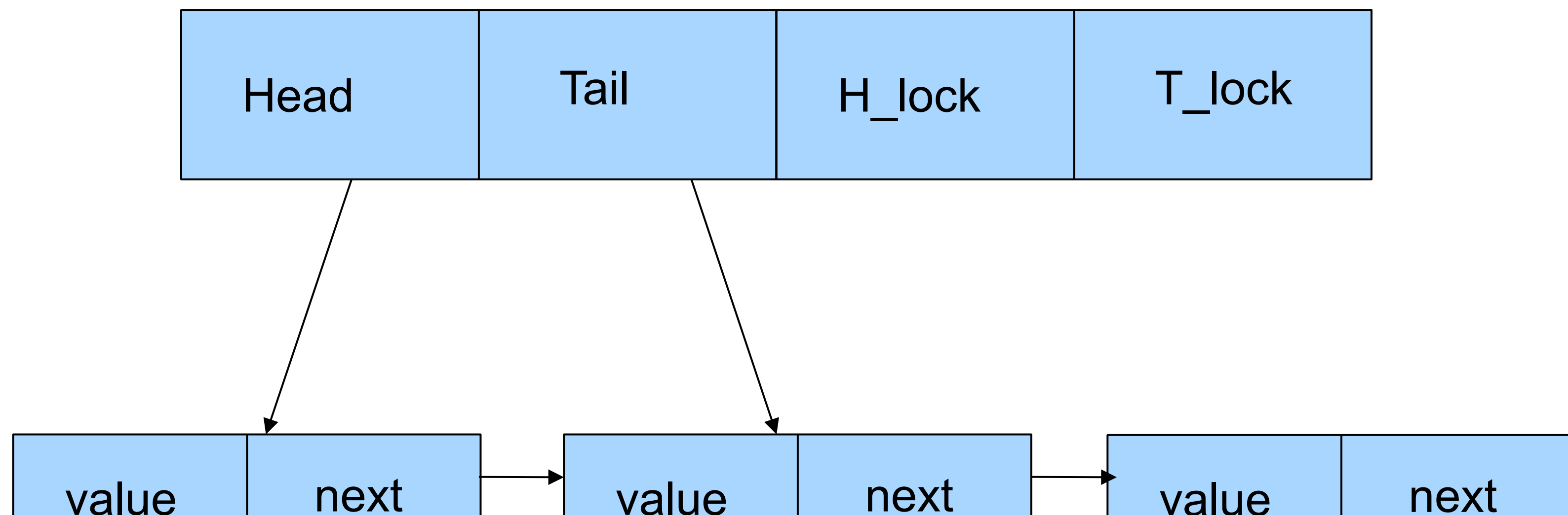
Queue Correctness Properties

1. The linked list is always connected
2. Nodes are only inserted after the last node in the linked list
3. Nodes are only deleted from the beginning of the linked list
4. Head always points to the first node in the list
5. Tail always points to a node in the list

Tricky Problems With Queues

- Tail lag
- Freeing dequeued nodes

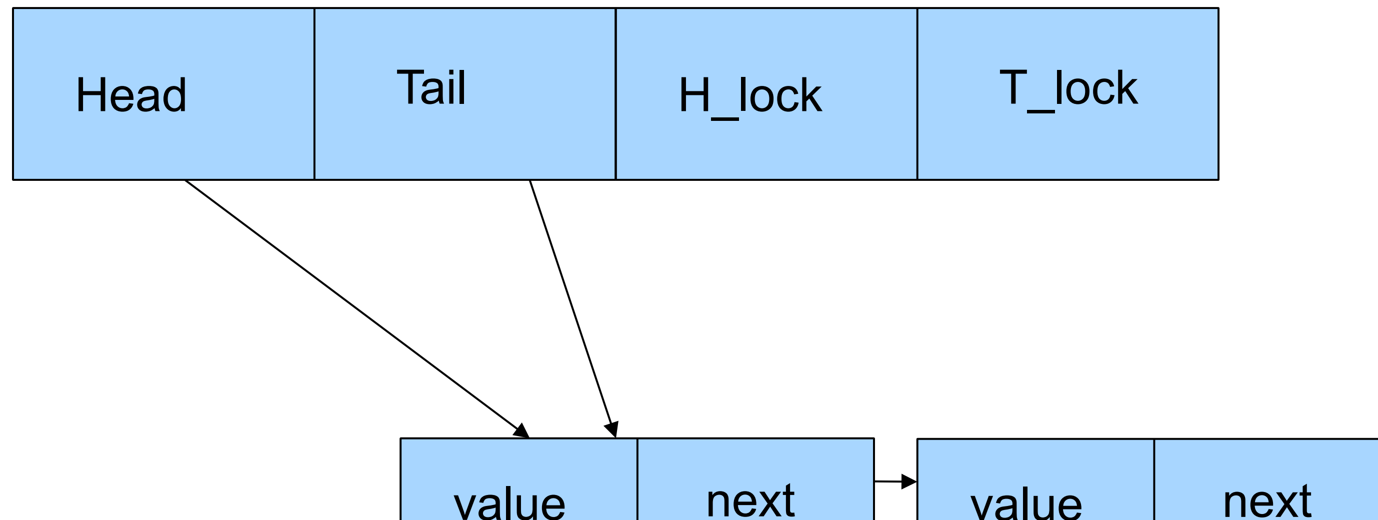
Problem – Tail Lag Behind Head



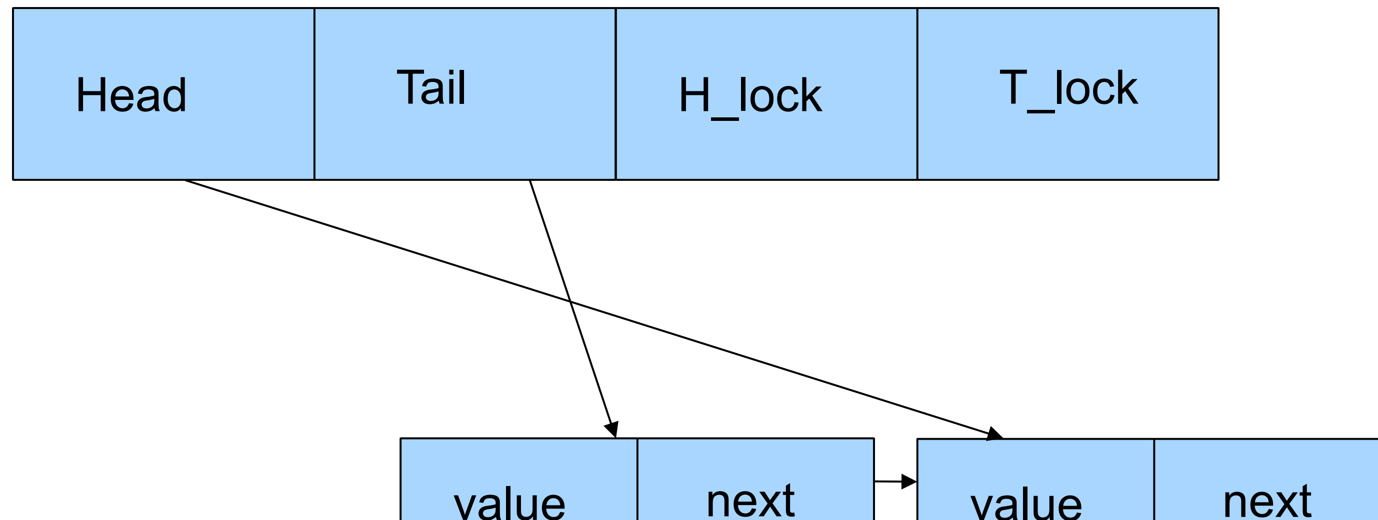
- Separate head and tail pointers
- Dequeueing threads can move the head pointer past the tail pointer if enqueueing threads are slow to update it

Portland State University

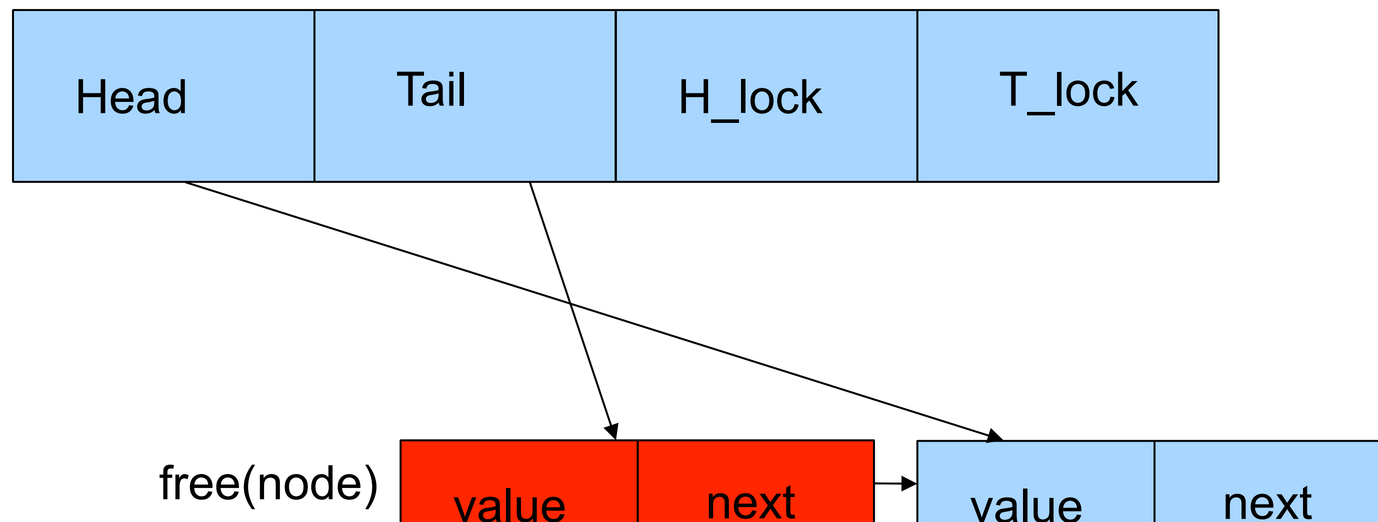
Problem – Tail Lag Behind Head



Problem – Tail Lag Behind Head



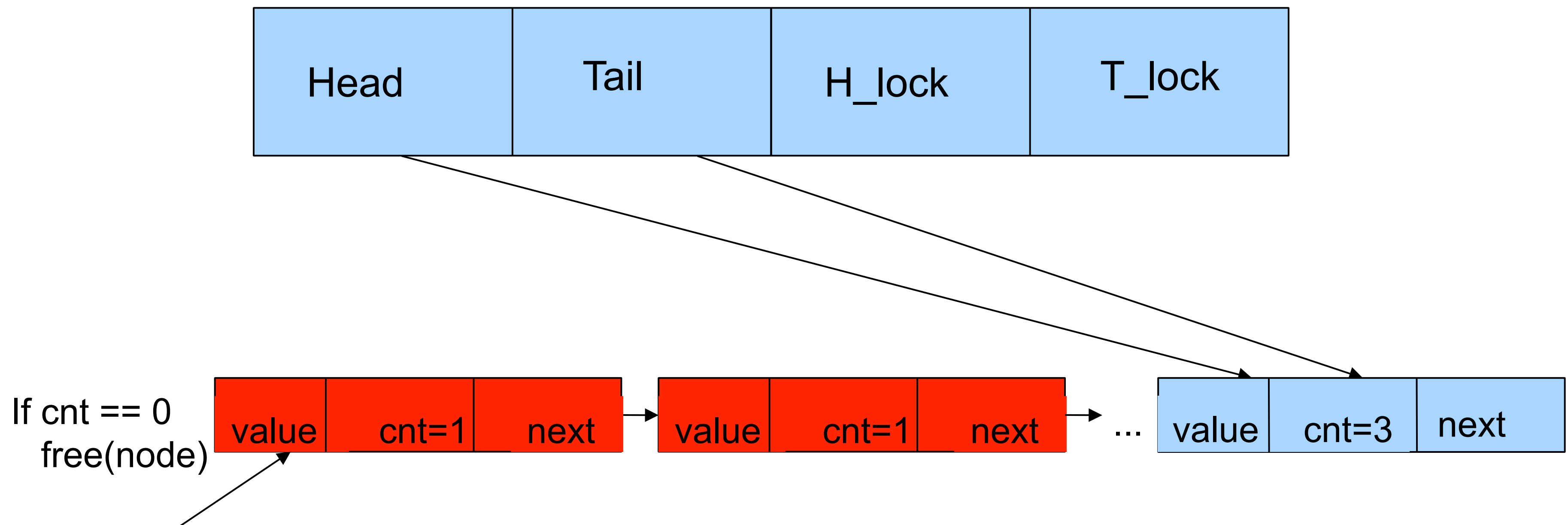
Problem – Tail Lag Behind Head



- The tail can end up pointing to freed memory!
- Can a dequeue help a slow enqueue by swinging the tail pointer before dequeuing the node
  - Will helping cause contention?
  - When is it safe to free memory?

Portland State University

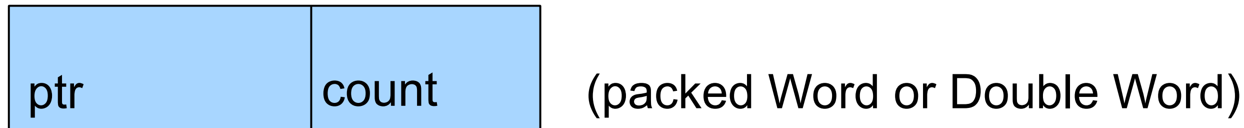
Problem With Reference Counting



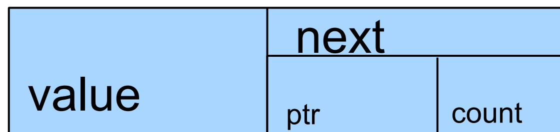
A slow process that never releases the reference count lock can cause the system to run out of memory

Non-Blocking Queue Algorithm

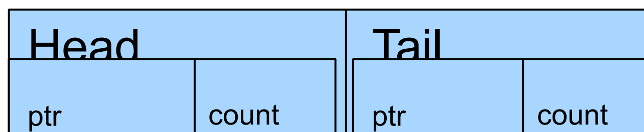
structure pointer_t {ptr: **pointer to** node t, count: **unsigned integer}**



structure node_t {value: data type, next: pointer_t}

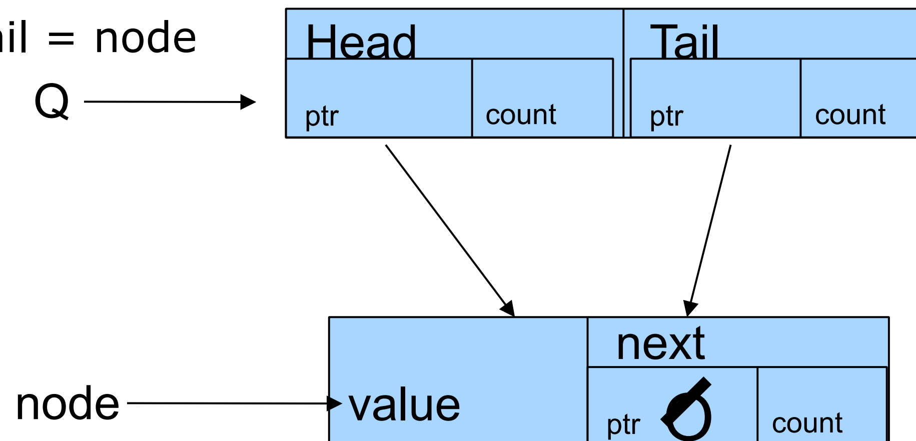


structure queue_t {Head: pointer_t, Tail: pointer_t}



Initialization

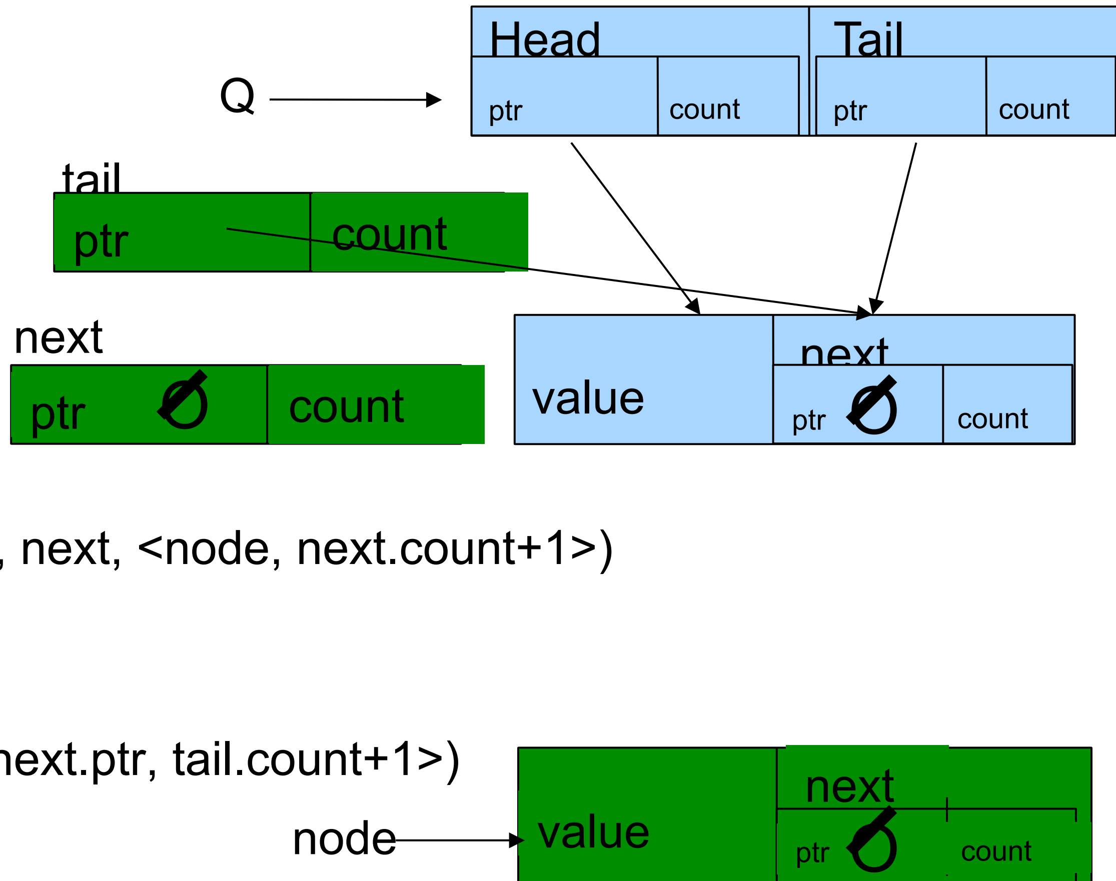
initialize(Q: **pointer to** queue_t)
node = new node()
node–>next.ptr = NULL
Q–>Head = Q–>Tail = node



Enqueue – Copy

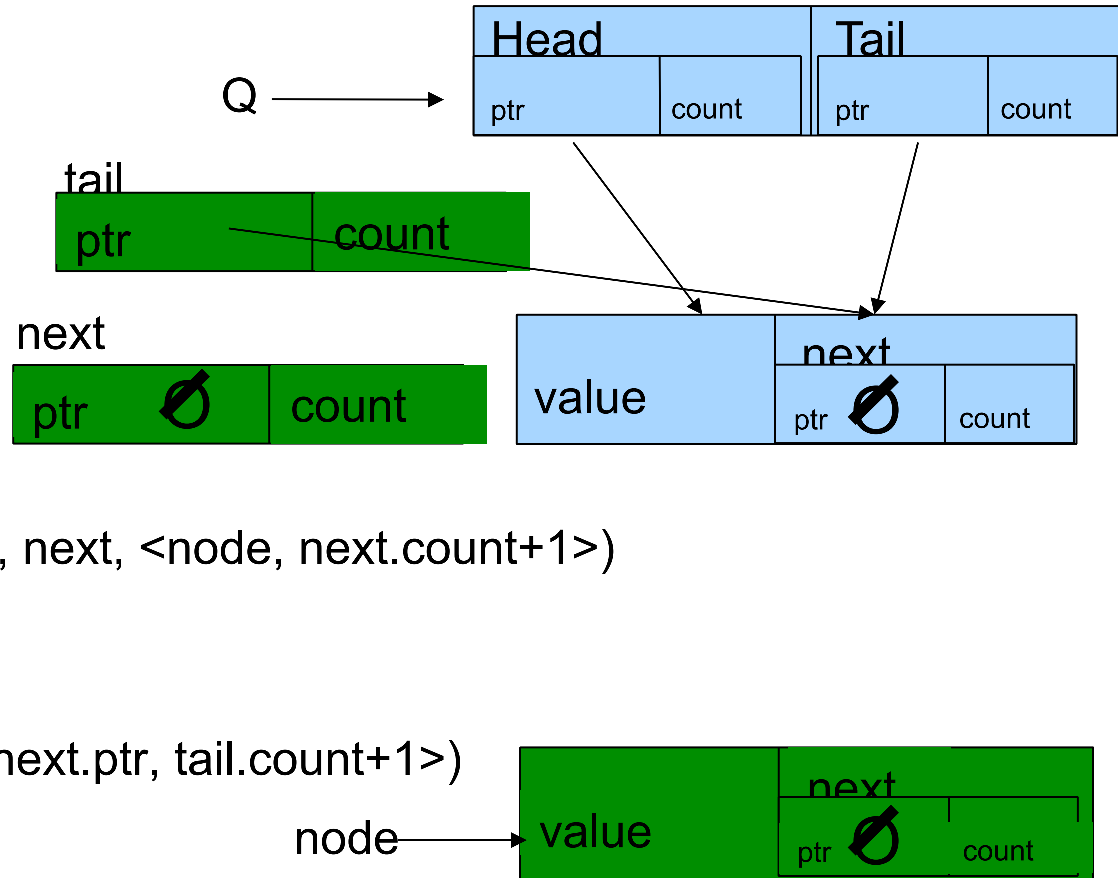
enqueue(Q: **pointer to** queue t, value: data type)
node = new node()
node–>value = value
node–>next.ptr = NULL

loop
  tail = Q–>Tail
  next = tail.ptr–>next
  if tail == Q–>Tail
    if next.ptr == NULL
      if CAS(&tail.ptr–>next, next, <node, next.count+1>)
        break
      endif
    else
      CAS(&Q–>Tail, tail, <next.ptr, tail.count+1>)
    endif
  endif
endloop
CAS(&Q–>Tail, tail, <node, tail.count+1>)



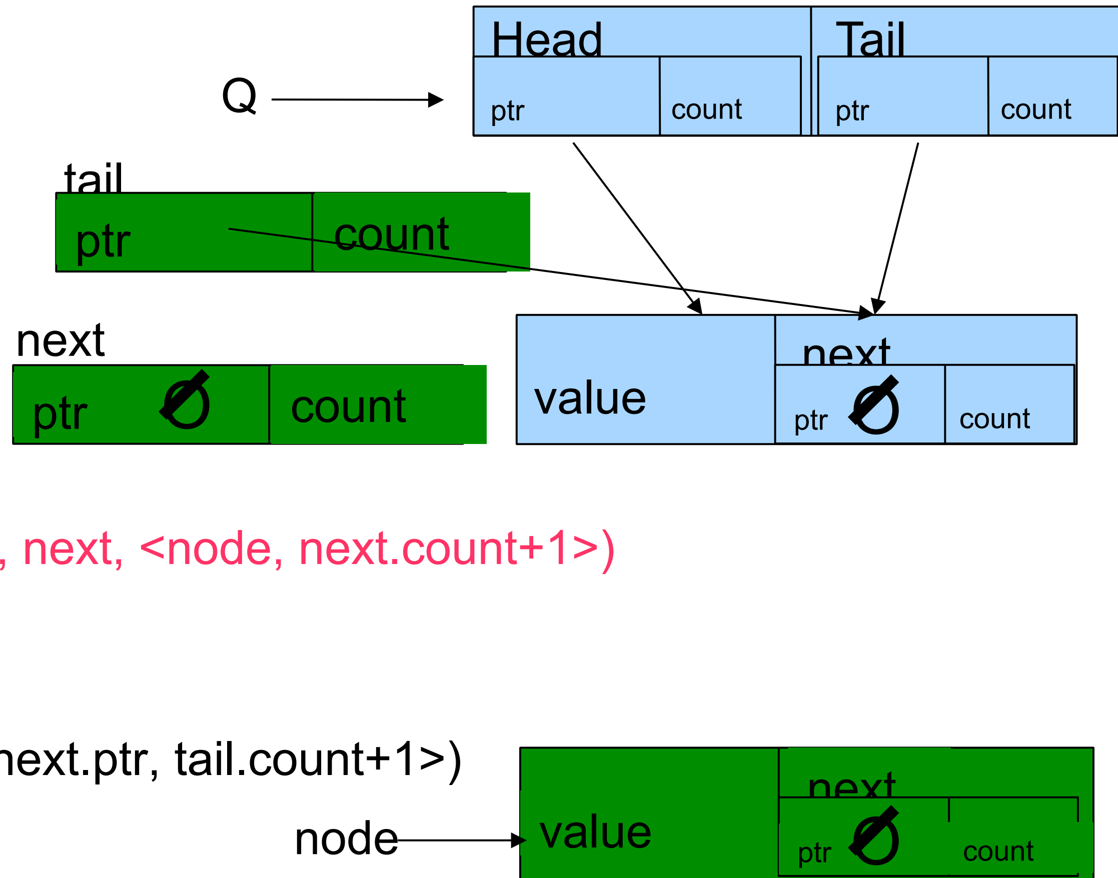
Enqueue - Check

```
enqueue(Q: pointer to queue t, value: data type)
node = new node()
node–>value = value
node–>next.ptr = NULL

loop
  tail = Q–>Tail
  next = tail.ptr–>next
  if tail == Q–>Tail
    if next.ptr == NULL
      if CAS(&tail.ptr–>next, next, <node, next.count+1>)
        break
      endif
    else
      CAS(&Q–>Tail, tail, <next.ptr, tail.count+1>)
    endif
  endif
endloop
CAS(&Q–>Tail, tail, <node, tail.count+1>)
```



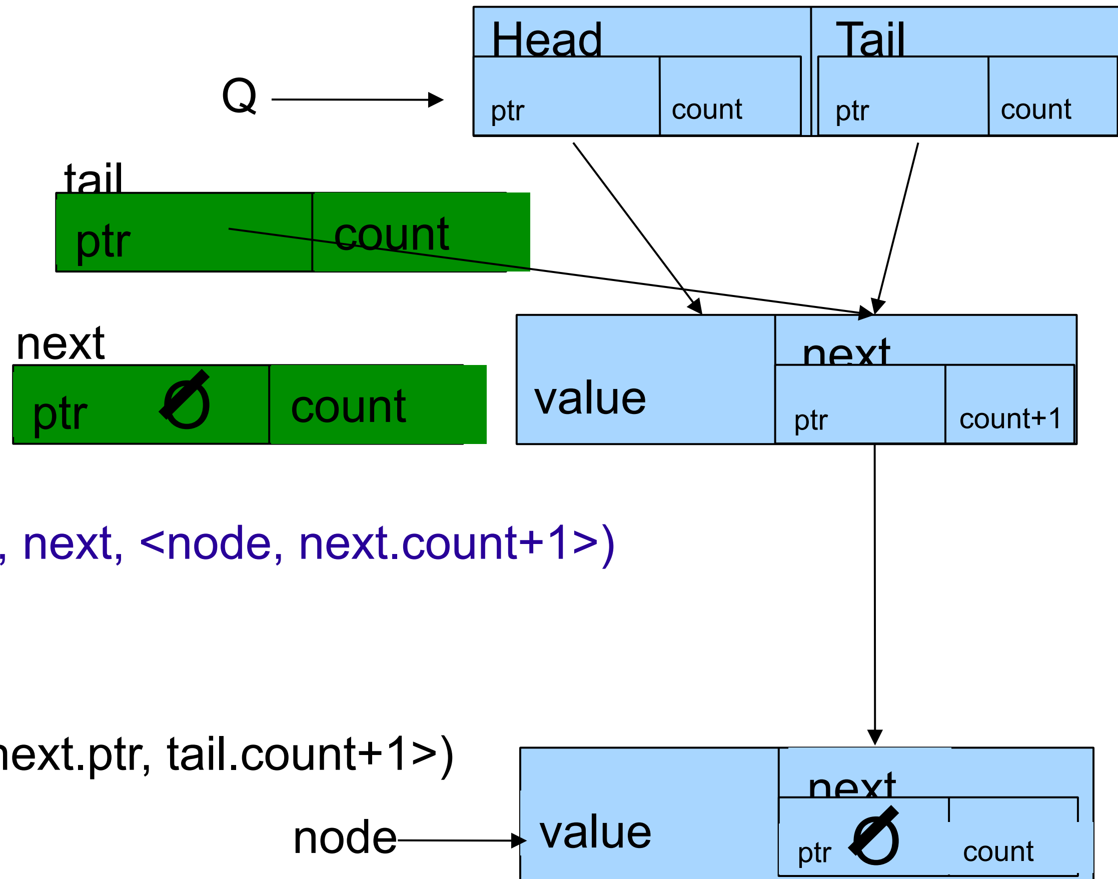
Enqueue – Try Commit

```
enqueue(Q: pointer to queue t, value: data type)
node = new node()
node–>value = value
node–>next.ptr = NULL

loop
   tail = Q–>Tail
   next = tail.ptr–>next
   if tail == Q–>Tail
      if next.ptr == NULL
         if CAS(&tail.ptr–>next, next, <node, next.count+1>)
            break
         endif
      else
         CAS(&Q–>Tail, tail, <next.ptr, tail.count+1>)
      endif
   endif
endloop
CAS(&Q–>Tail, tail, <node, tail.count+1>)
```



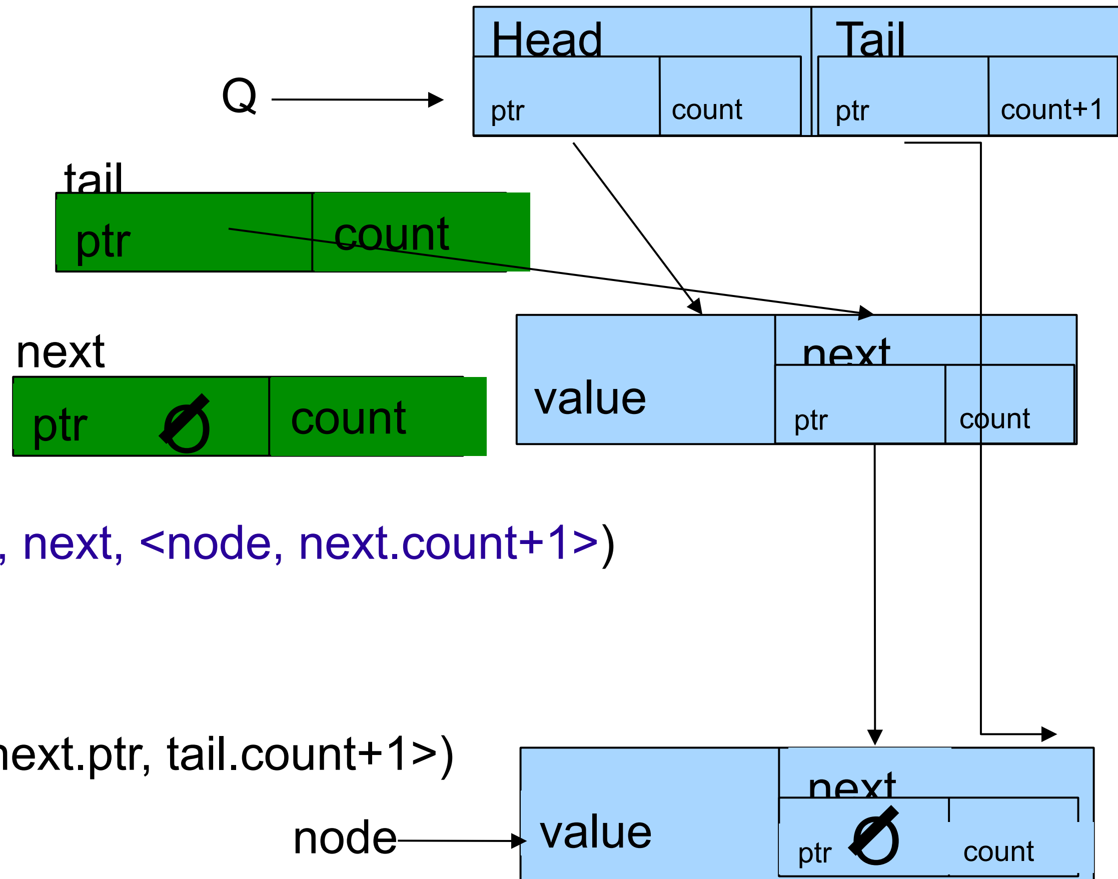
Enqueue - Succeed

```
enqueue(Q: pointer to queue t, value: data type)
node = new node()
node–>value = value
node–>next.ptr = NULL


loop
   tail = Q–>Tail
   next = tail.ptr–>next
   if tail == Q–>Tail
      if next.ptr == NULL
         if CAS(&tail.ptr–>next, next, <node, next.count+1>)
            break
         endif
      else
         CAS(&Q–>Tail, tail, <next.ptr, tail.count+1>)
      endif
   endif
endloop
CAS(&Q–>Tail, tail, <node, tail.count+1>)
```



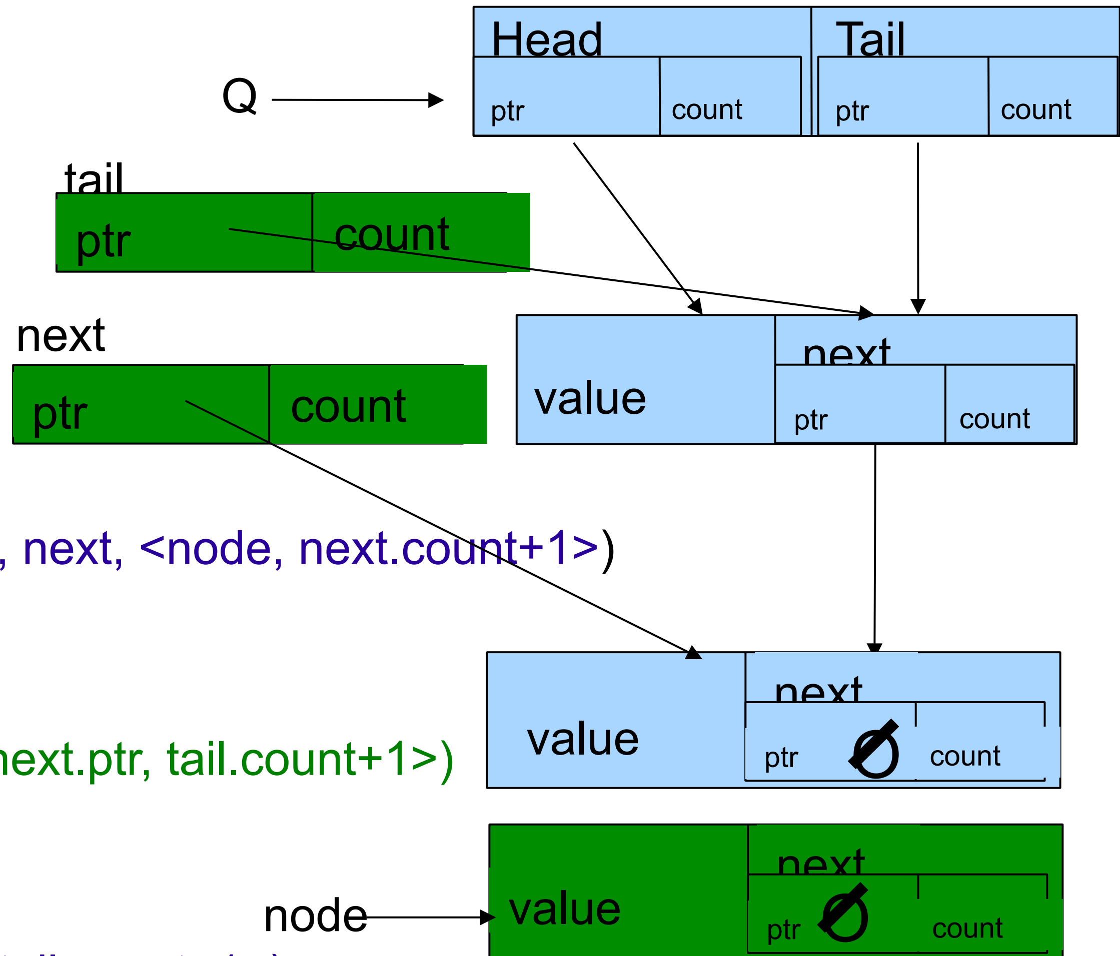
Enqueue - Swing Tail

```
enqueue(Q: pointer to queue t, value: data type)
node = new node()
node–>value = value
node–>next.ptr = NULL

loop
   tail = Q–>Tail
   next = tail.ptr–>next
   if tail == Q–>Tail
      if next.ptr == NULL
         if CAS(&tail.ptr–>next, next, <node, next.count+1>)
            break
         endif
      else
         CAS(&Q–>Tail, tail, <next.ptr, tail.count+1>)
      endif
   endif
endloop
CAS(&Q–>Tail, tail, <node, tail.count+1>)
```



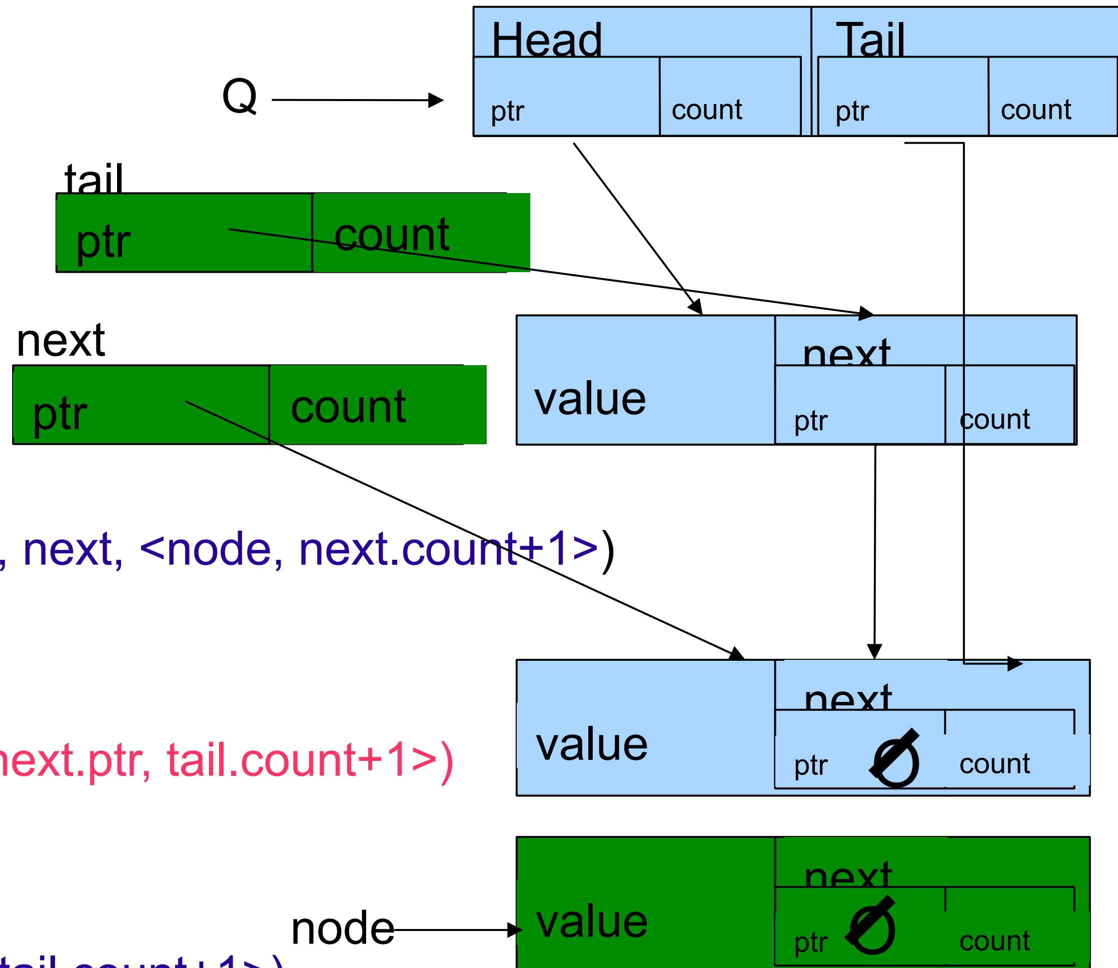
Enqueue – Other Process Won

```
enqueue(Q: pointer to queue t, value: data type)
node = new node()
node–>value = value
node–>next.ptr = NULL

loop
   tail = Q–>Tail
   next = tail.ptr–>next
   if tail == Q–>Tail
      if next.ptr == NULL
         if CAS(&tail.ptr–>next, next, <node, next.count+1>)
            break
         endif
      else
         CAS(&Q–>Tail, tail, <next.ptr, tail.count+1>)
      endif
   endif
endloop
CAS(&Q–>Tail, tail, <node, tail.count+1>)
```



Enqueue – Catch Up

```
enqueue(Q: pointer to queue t, value: data type)
node = new node()
node–>value = value
node–>next.ptr = NULL

loop
   tail = Q–>Tail
   next = tail.ptr–>next
   if tail == Q–>Tail
      if next.ptr == NULL
         if CAS(&tail.ptr–>next, next, <node, next.count+1>)
            break
         endif
      else
         CAS(&Q–>Tail, tail, <next.ptr, tail.count+1>)
      endif
   endif
endloop
CAS(&Q–>Tail, tail, <node, tail.count+1>)
```



Enqueue – Retry Commit

```
enqueue(Q: pointer to queue t, value: data type)
node = new node()
node–>value = value
node–>next.ptr = NULL

loop
   tail = Q–>Tail
   next = tail.ptr–>next
   if tail == Q–>Tail
      if next.ptr == NULL
         if CAS(&tail.ptr–>next, next, <node, next.count+1>)
            break
         endif
      else
         CAS(&Q–>Tail, tail, <next.ptr, tail.count+1>)
      endif
   endif
endloop
CAS(&Q–>Tail, tail, <node, tail.count+1>)
```

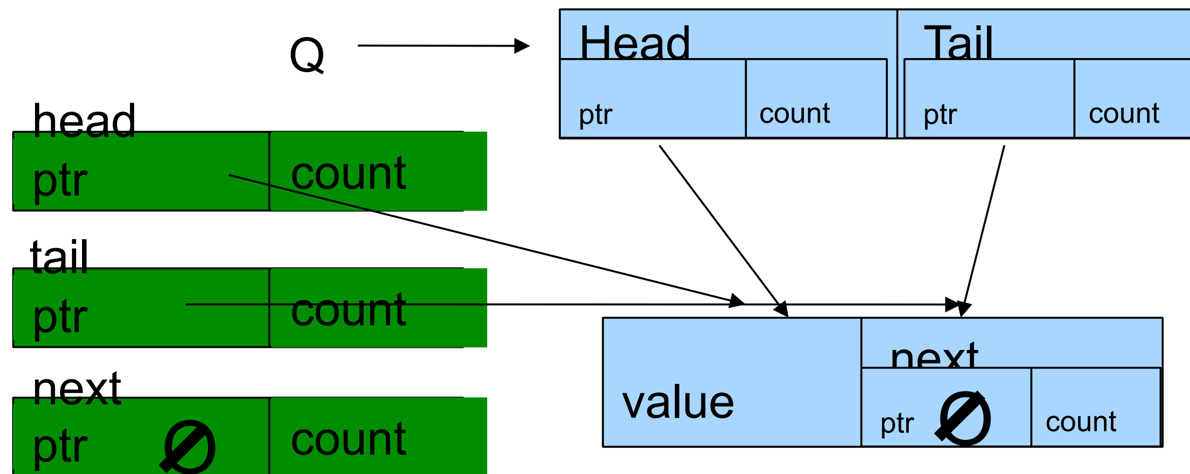
Q
Head
ptr
count
Tail
ptr
count
tail
ptr
count
next
ptr
count
value
next
ptr
count
value
next
ptr
coun t+1
node
value
next
ptr
count

Enqueue

- When does the enqueue *take effect?*
- Any thoughts?

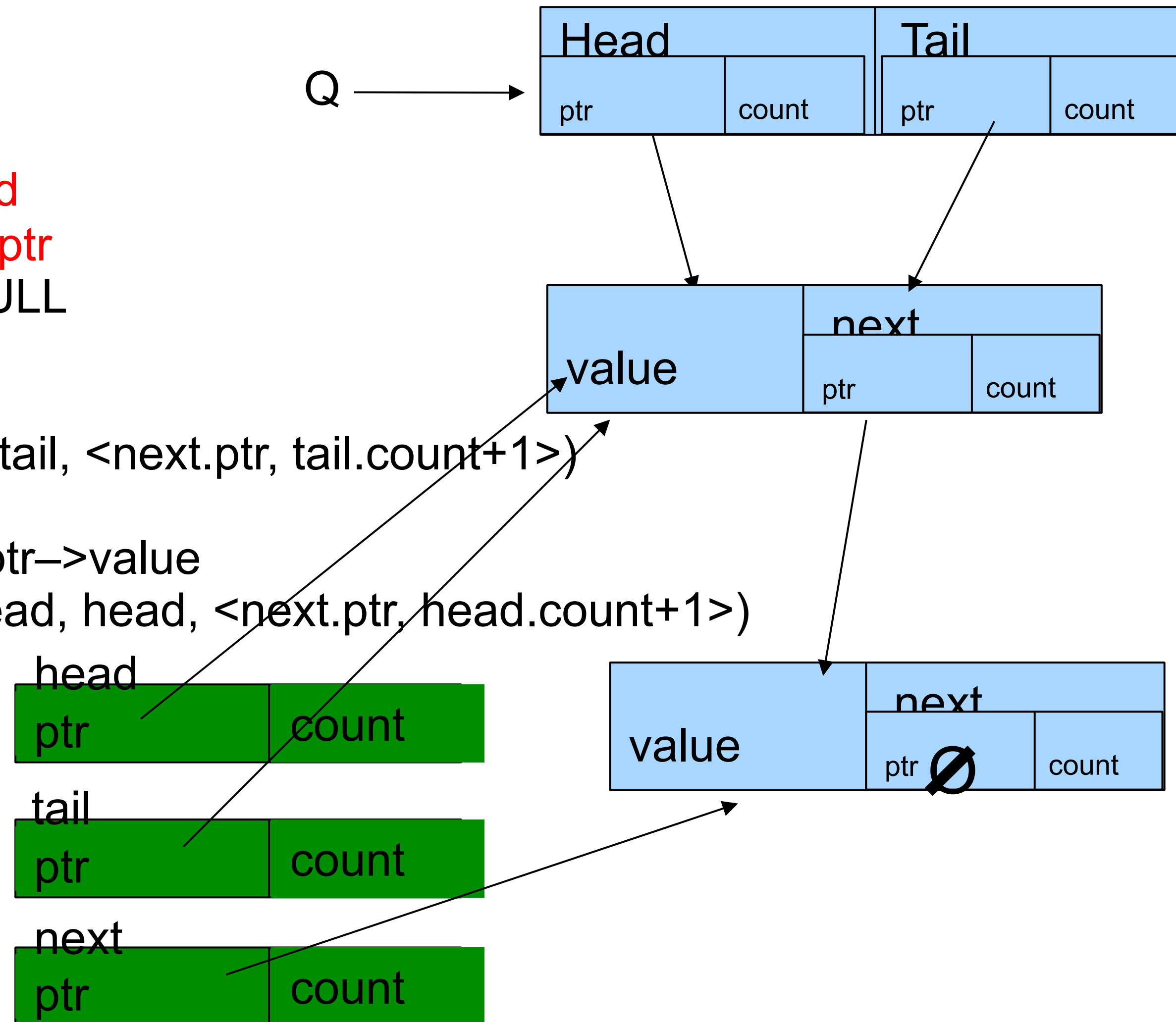
Dequeue – Copy

```
dequeue(Q: pointer to queue_t, pvalue: pointer to data type): boolean
loop
   head = Q–>Head
   tail = Q–>Tail
   next = head–>next
   if head == Q–>Head
      if head.ptr == tail.ptr
         if next.ptr == NULL
            return FALSE
         endif
```



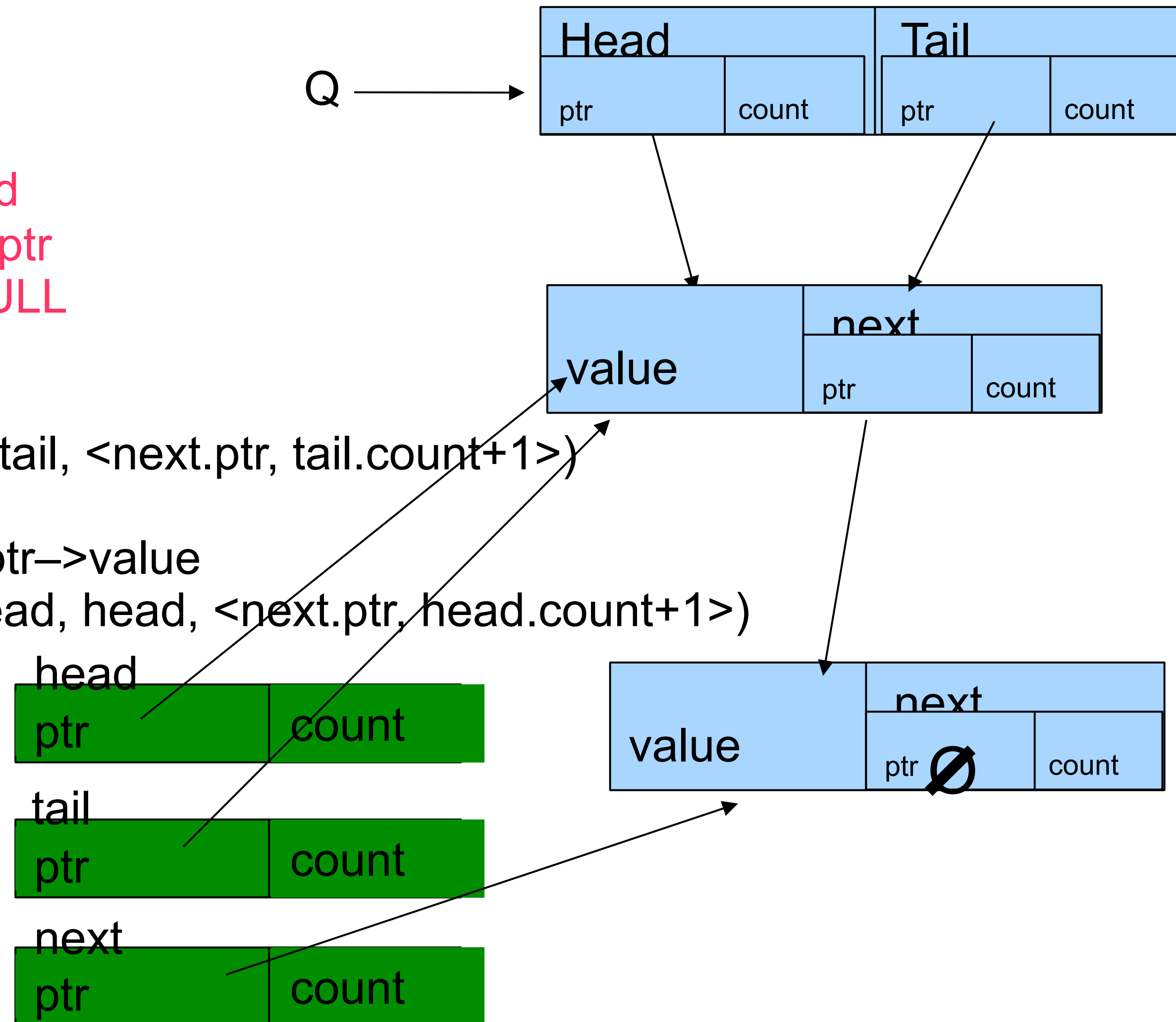
Dequeue – Check

dequeue(Q: **pointer to** queue_t, pvalue: **pointer to** data type): **boolean**
loop
  head = Q–>Head
  tail = Q–>Tail
  next = head–>next
  if head == Q–>Head
    if head.ptr == tail.ptr
      if next.ptr == NULL
        return FALSE
      endif
      CAS(&Q–>Tail, tail, <next.ptr, tail.count+1>)
    else
      *pvalue = next.ptr–>value
      if CAS(&Q–>Head, head, <next.ptr, head.count+1>)
        break
      endif
    endif
  endif
endloop
free(head.ptr)
return TRUE



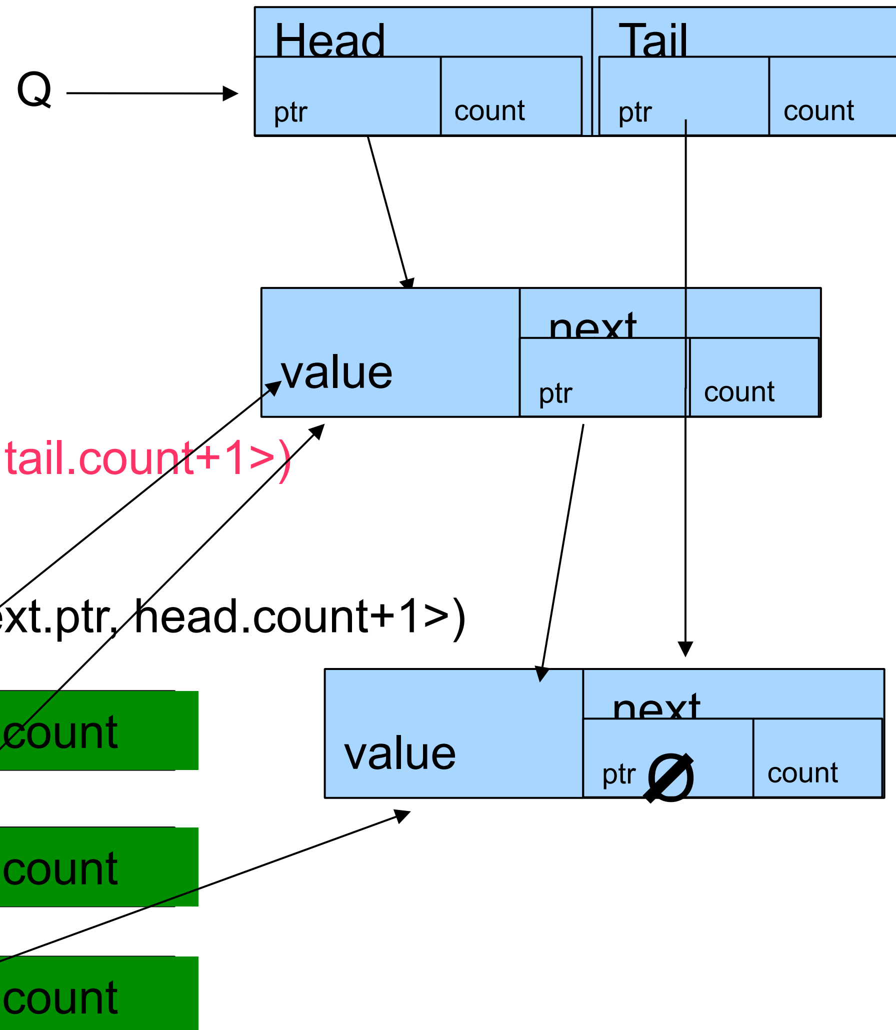
Dequeue – Empty Queue

```
dequeue(Q: pointer to queue_t, pvalue: pointer to data type): boolean
loop
   head = Q–>Head
   tail = Q–>Tail
   next = head–>next
   if head == Q–>Head
      if head.ptr == tail.ptr
         if next.ptr == NULL
            return FALSE
         endif
         CAS(&Q–>Tail, tail, <next.ptr, tail.count+1>)
      else
         *pvalue = next.ptr–>value
         if CAS(&Q–>Head, head, <next.ptr, head.count+1>)
            break
         endif
      endif
   endif
endloop
free(head.ptr)
return TRUE
```



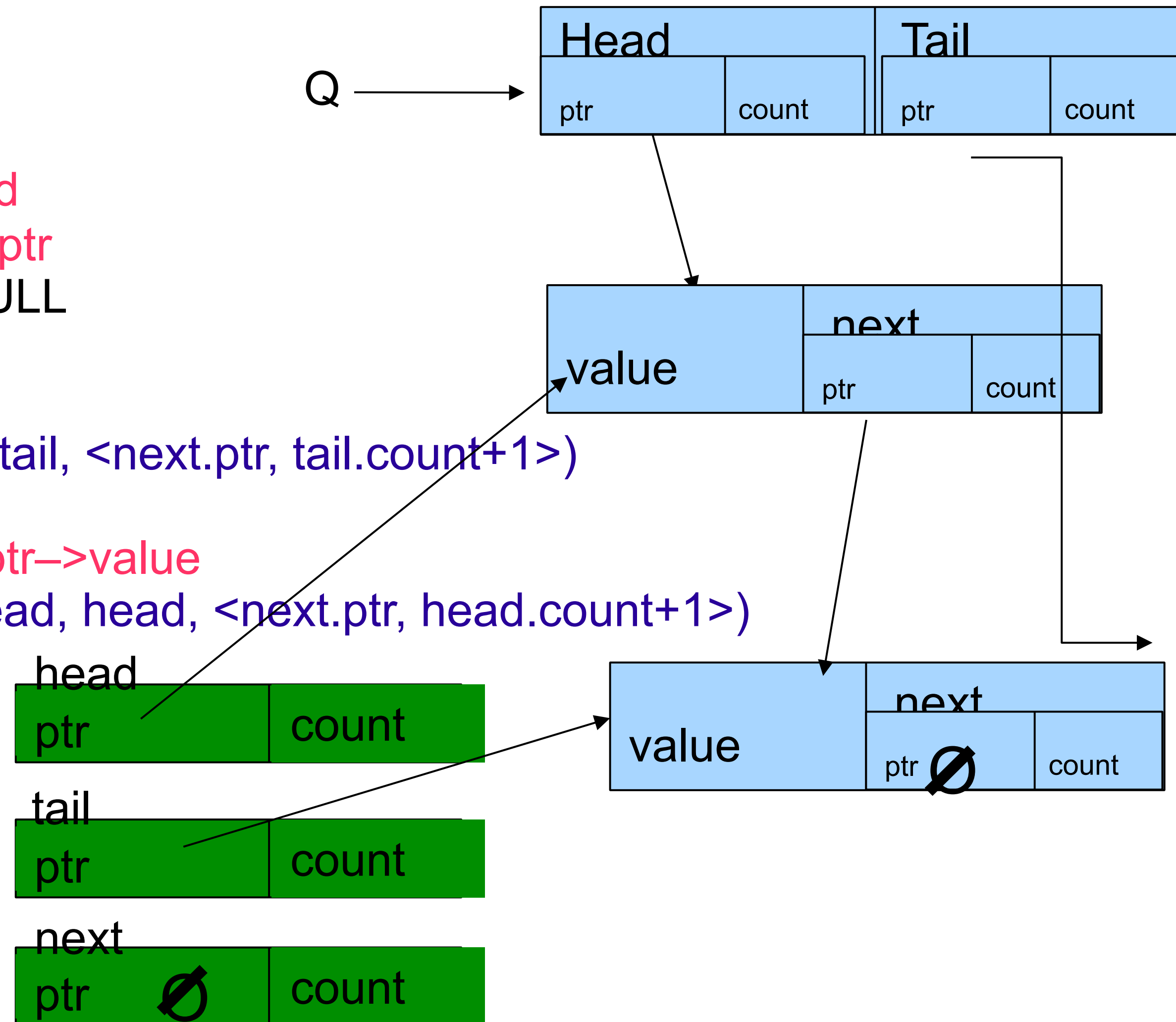
Dequeue – Help Tail Catch Up

```
dequeue(Q: pointer to queue_t, pvalue: pointer to data type): boolean
loop
  head = Q–>Head
  tail = Q–>Tail
  next = head–>next
  if head == Q–>Head
    if head.ptr == tail.ptr
      if next.ptr == NULL
        return FALSE
      endif
      CAS(&Q–>Tail, tail, <next.ptr, tail.count+1>)
    else
      *pvalue = next.ptr–>value
      if CAS(&Q–>Head, head, <next.ptr, head.count+1>)
        break
      endif
    endif
  endif
endloop
free(head.ptr)
return TRUE
```



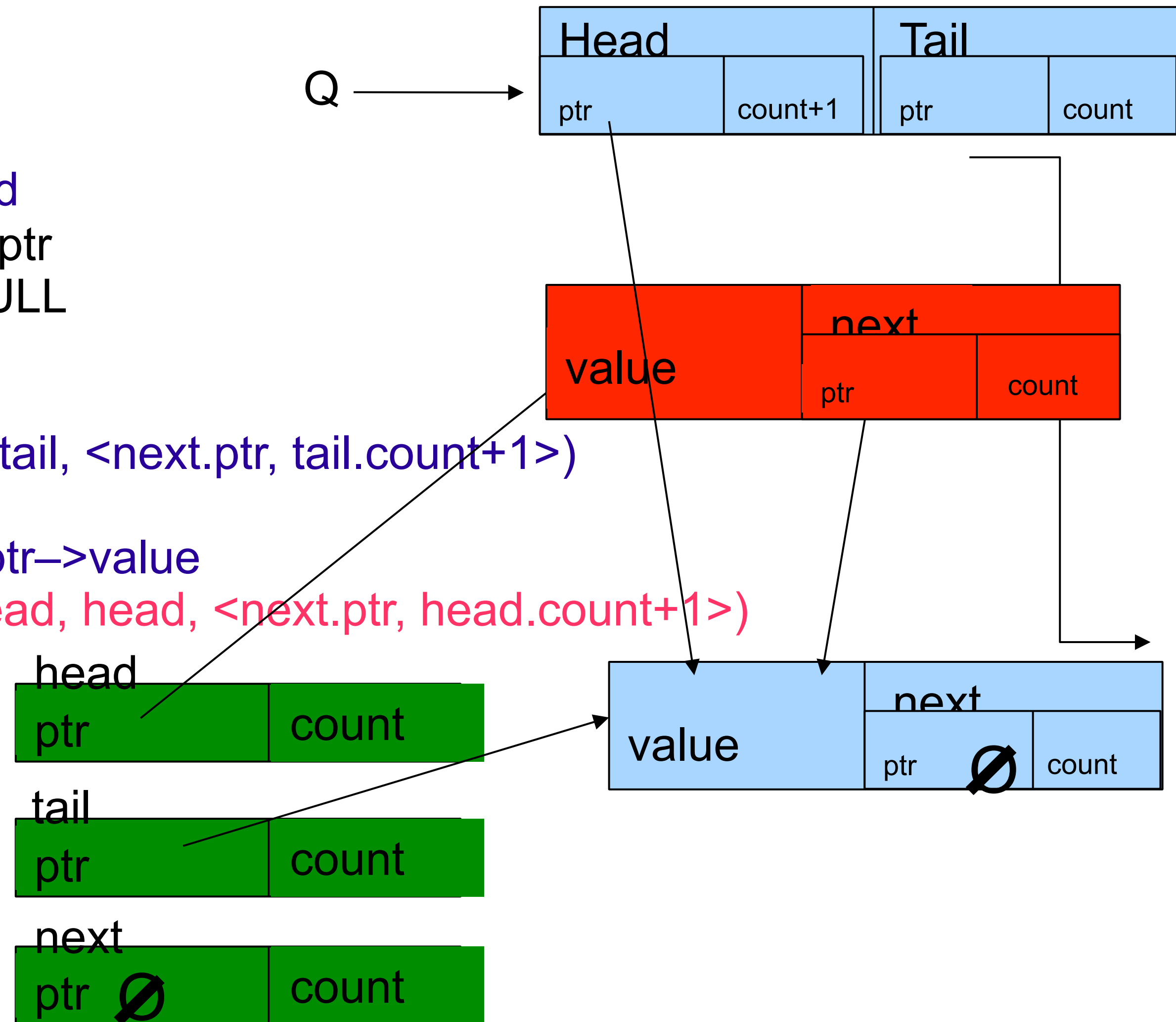
Dequeue – Grab Node Value

```
dequeue(Q: pointer to queue_t, pvalue: pointer to data type): boolean
loop
   head = Q–>Head
   tail = Q–>Tail
   next = head–>next
   if head == Q–>Head
      if head.ptr == tail.ptr
         if next.ptr == NULL
            return FALSE
         endif
         CAS(&Q–>Tail, tail, <next.ptr, tail.count+1>)
      else
         *pvalue = next.ptr–>value
         if CAS(&Q–>Head, head, <next.ptr, head.count+1>)
            break
         endif
      endif
   endif
endloop
free(head.ptr)
return TRUE
```



Dequeue - Commit and Free Node

```
dequeue(Q: pointer to queue_t, pvalue: pointer to data type): boolean
loop
   head = Q–>Head
   tail = Q–>Tail
   next = head–>next
   if head == Q–>Head
      if head.ptr == tail.ptr
         if next.ptr == NULL
            return FALSE
         endif
         CAS(&Q–>Tail, tail, <next.ptr, tail.count+1>)
      else
         *pvalue = next.ptr–>value
         if CAS(&Q–>Head, head, <next.ptr, head.count+1>)
            break
         endif
      endif
   endif
endloop
free(head.ptr)
return TRUE
```



Dequeue

- When does the dequeue *take effect*
- Any thoughts?

Two-Lock 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}

Blocking Using Two Locks

One for tail pointer updates and a second for head pointer updates

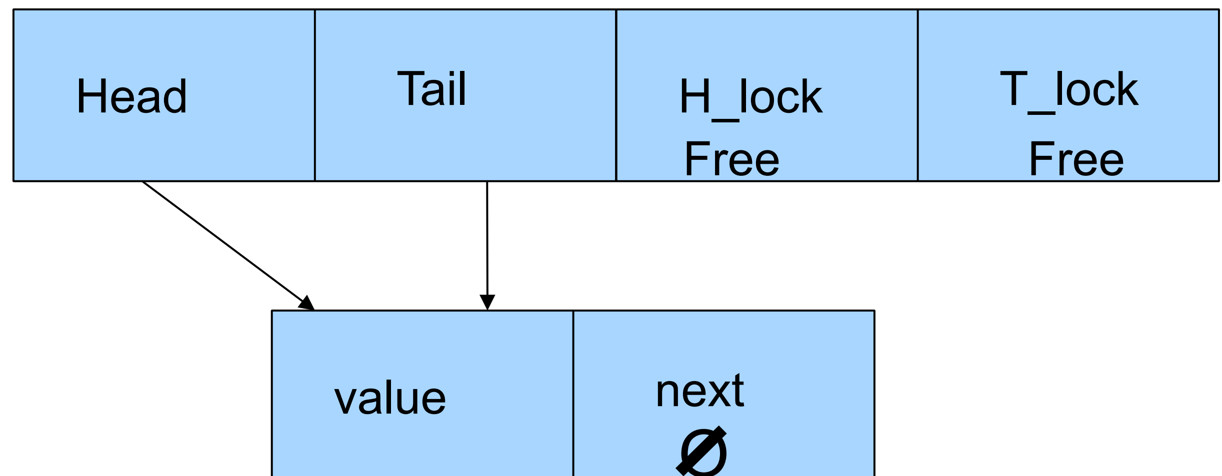
- Enqueue process locks tail pointer
- Dequeue process locks head pointer

A dummy node prevents enqueue from needing to access the head pointer and dequeue from needing to access the tail pointer

- allows concurrent enqueues and dequeues

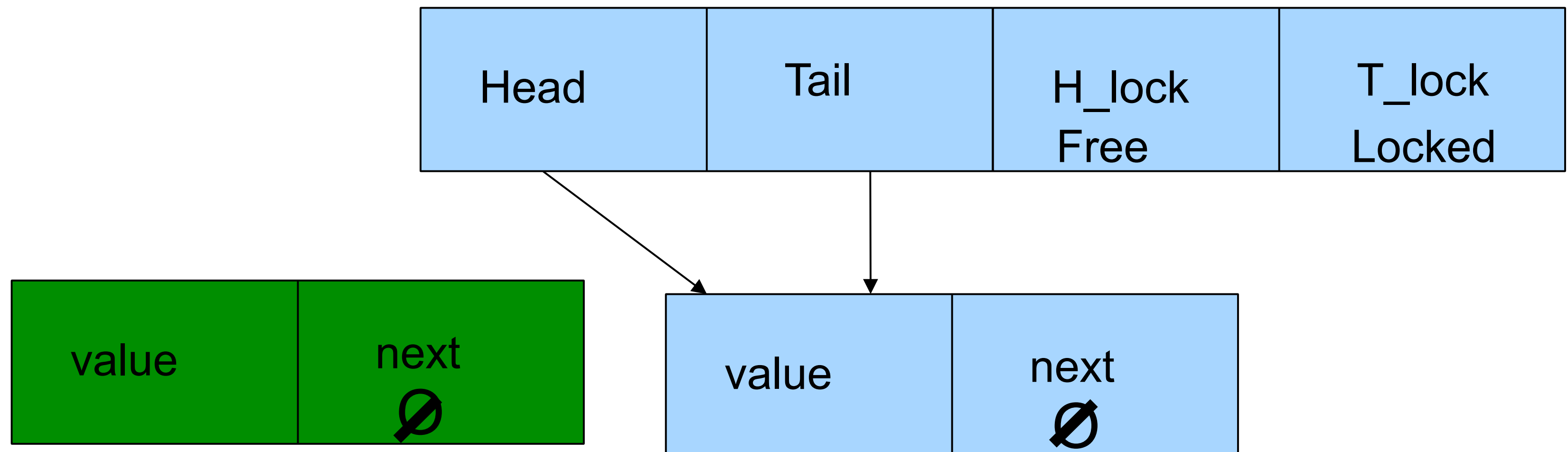
Algorithm – Two-Lock Queue

```
initialize(Q: pointer to queue_t)
    node = new_node()
    node->next = NULL
    Q->Head = Q->Tail = node
    Q->H_lock = Q->T_lock = FREE
```



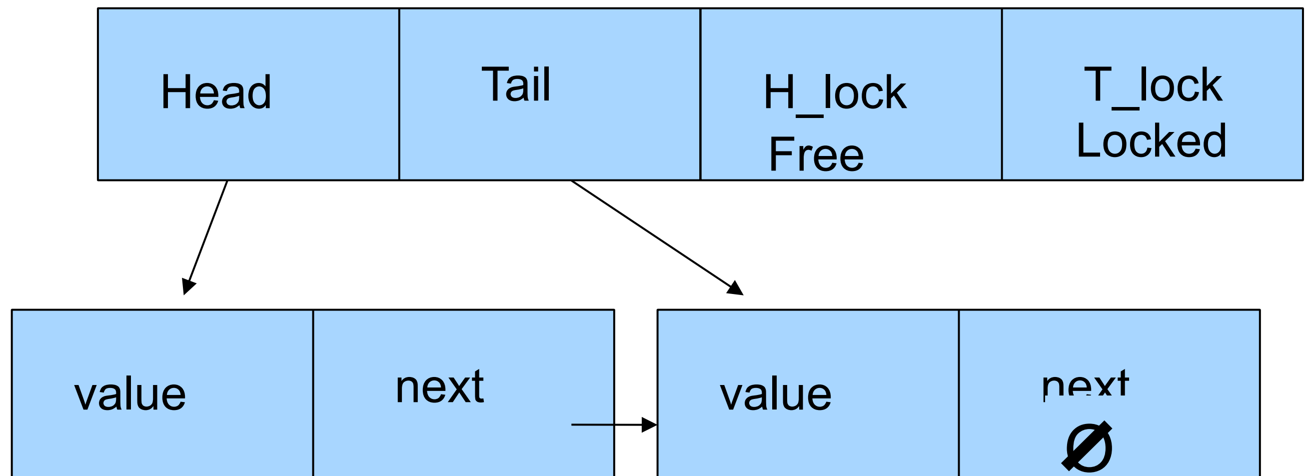
Enqueue

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



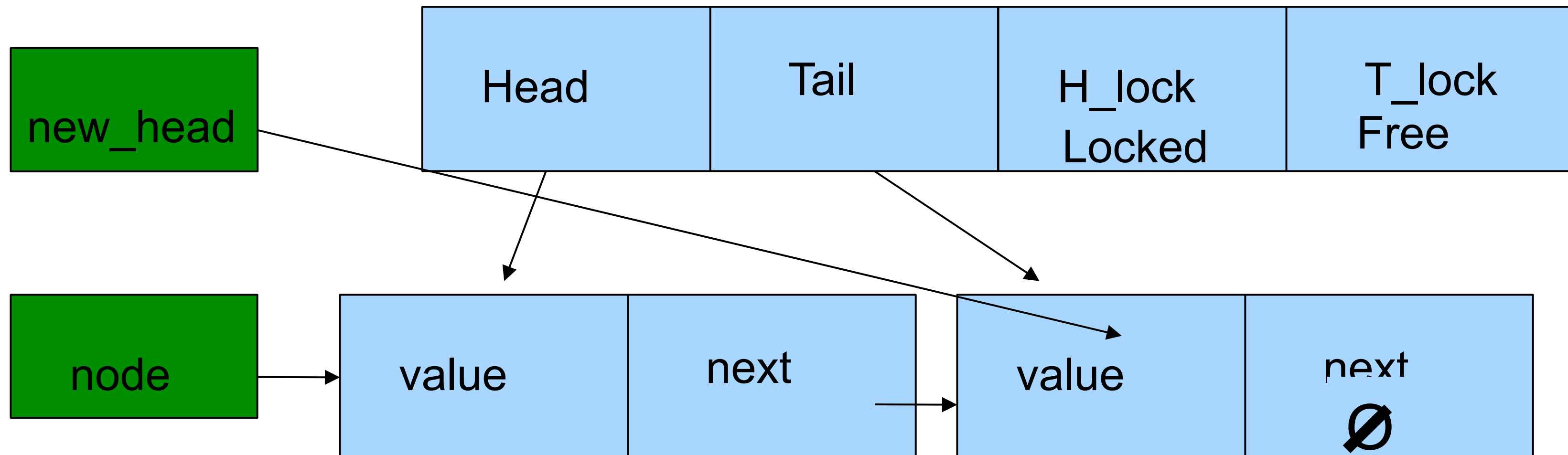
Enqueue

```
enqueue(Q: pointer to queue_t, value: data type)
    node = new_node()            // Allocate a new node from the free list
    node->value = value          // Write value to be enqueued into node
    node->next = NULL            // Set node's next pointer 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 point to node
    unlock(&Q->T_lock)           // Release T_lock
```



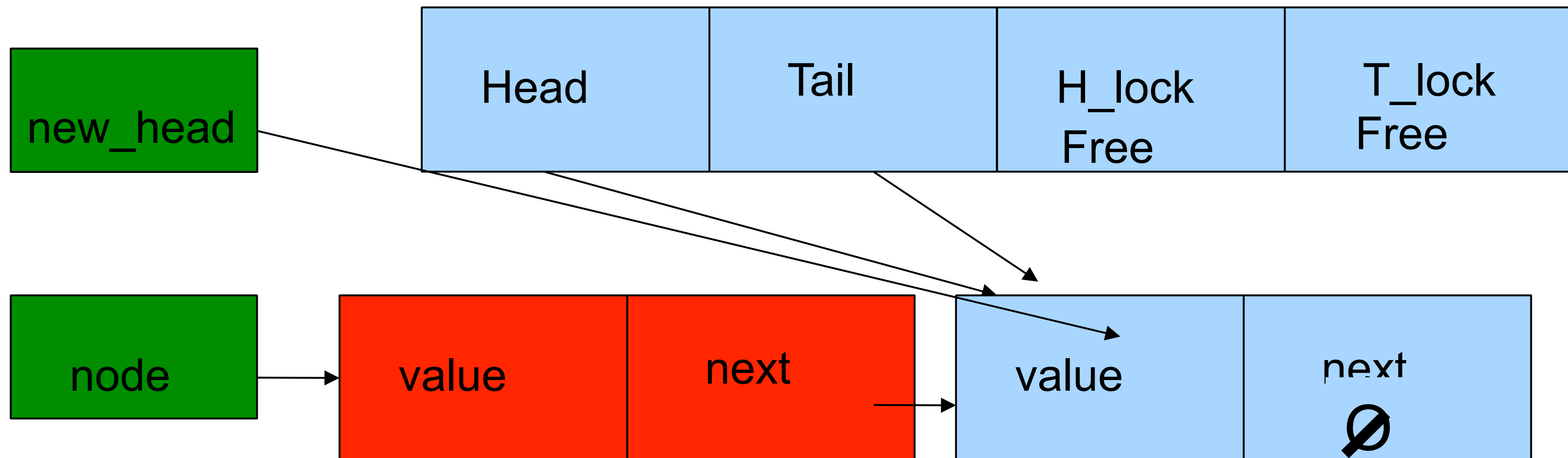
Dequeue

```
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
```



Dequeue

```
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
```



Algorithm – Two-Lock

- There is one case where 2 threads can access the same node at the same time
  - When is it?
  - Why is it not a problem?
  - Any other thoughts?

Performance Measurements

- The code was heavily optimized
- The test threads alternatively enqueue and dequeue items with *other work* (busy waiting) in between
- Multiple threads run on each CPU to simulate multi-programming

Dedicated Processor

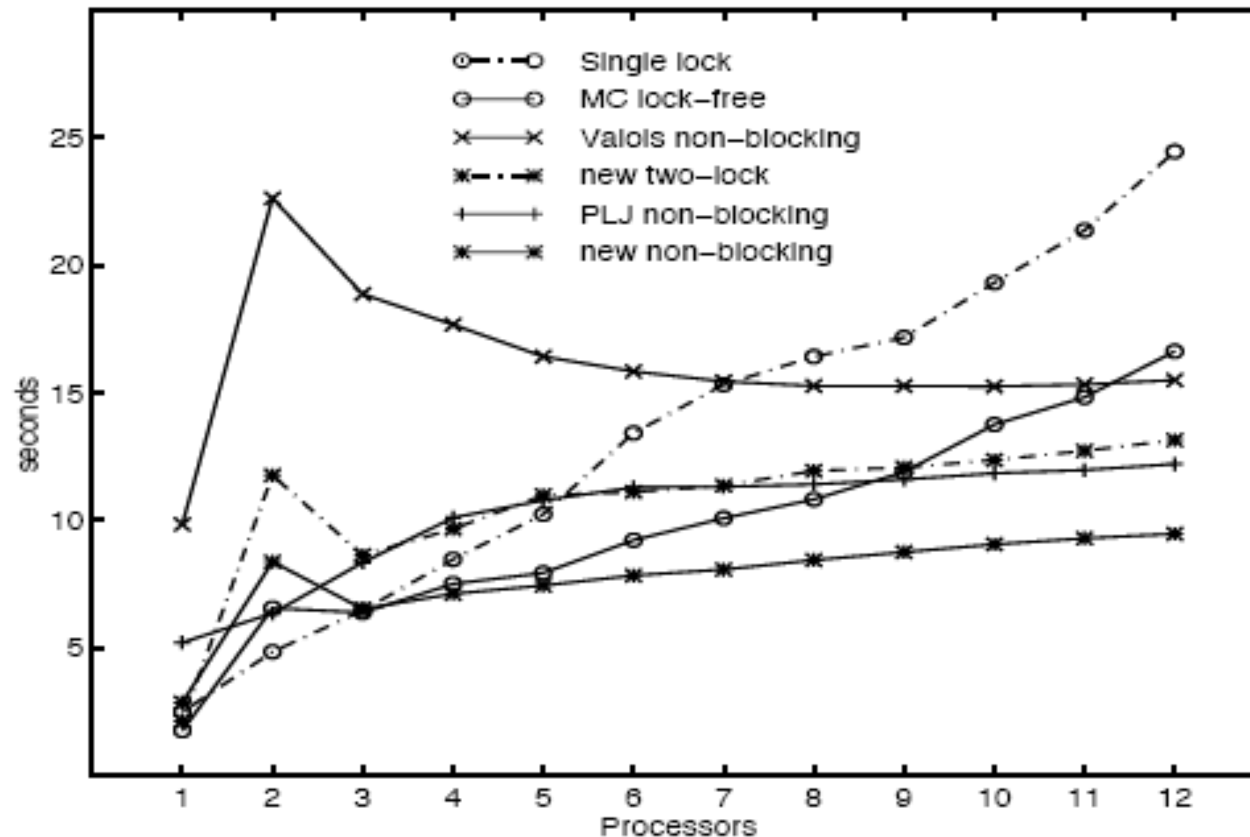


Figure 3: Net execution time for one million enqueue/dequeue pairs on a dedicated multiprocessor.

Two Processes Per Processor

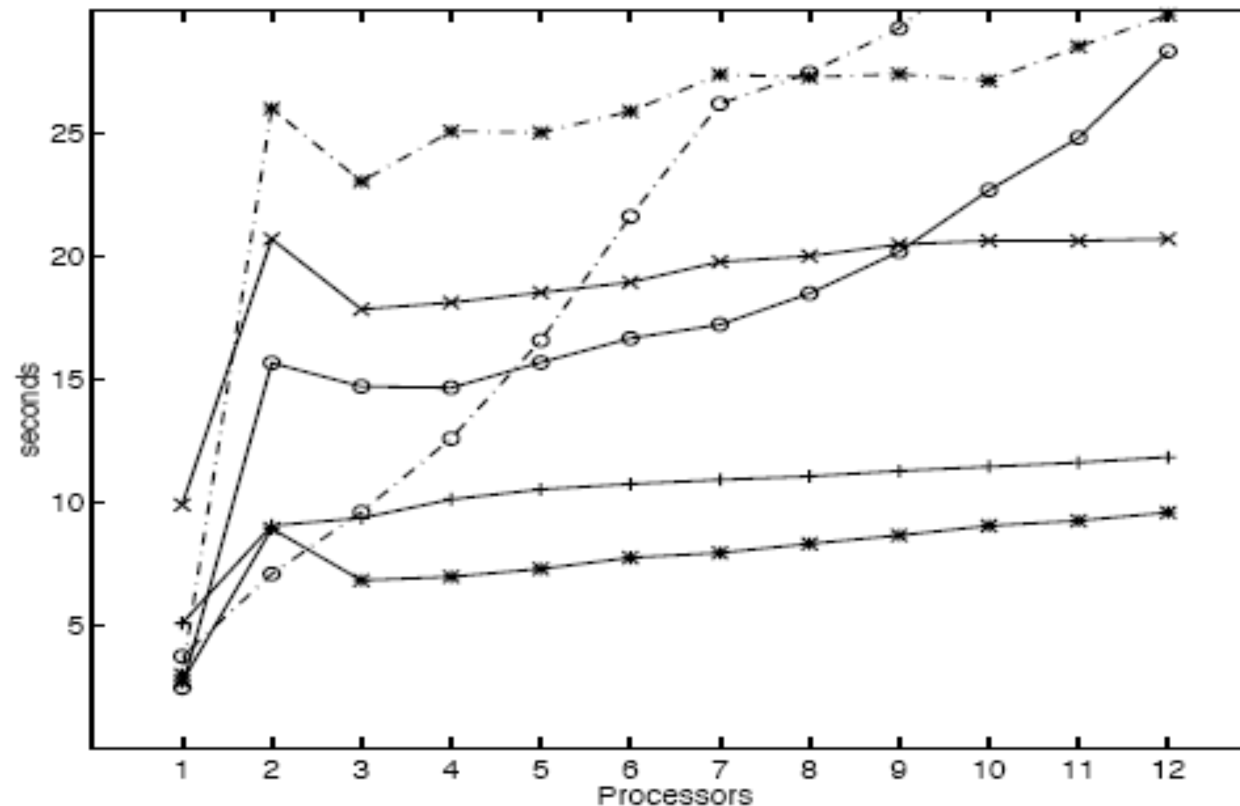


Figure 4: Net execution time for one million enqueue/dequeue pairs on a multiprogrammed system with 2 processes per processor.

Three Processes Per Processor

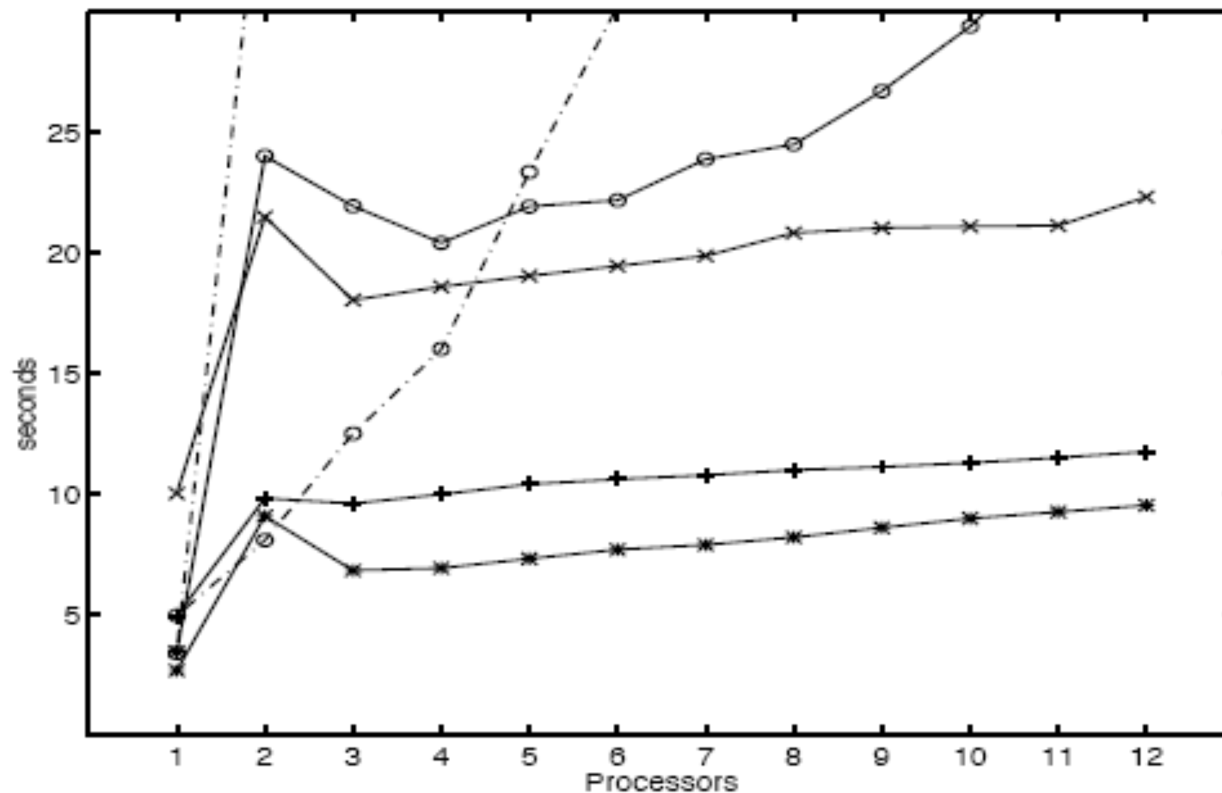


Figure 5: Net execution time for one million enqueue/dequeue pairs on a multiprogrammed system with 3 processes per processor.

Conclusion

- Non-blocking synchronization is the clean winner for multi-programmed multiprocessor systems
- Above 5 processors, the new non-blocking queue is the best
- For hardware that only supports test-and-set, use the two lock queue
- For two or fewer processors use a single lock algorithm for queues because overhead of more complicated algorithms outweighs the gains