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

Maged M. Michael & Michael L. Scott

Presented by
Ahmed Badran

Agenda

- Terminology
- Blocking vs. Non-Blocking algorithms
- Algorithm walkthroughs
- Comparison to previous algorithms
- Performance

Terminology (1)

- Blocking algorithm
  - Uses locks
  - May deadlock
  - Processes may wait for arbitrarily long times
  - Lock/unlock primitives need to interact with scheduling logic to avoid priority inversion
  - Possibility of starvation

Terminology (2)

- Non-Blocking algorithm
  - One of many processes accessing the shared data is guaranteed to complete in a finite number of steps, may starve others
- Wait-free algorithm
  - All processes accessing the shared data structure are guaranteed to complete in a finite number of steps
  - Wait-free = Non-blocking + Starvation free

Terminology (3)

- Linearizable data structure (atomic?)

  A data structure gives an external observer the illusion that the operations takes effect instantaneously

  One would expect:

  if (enq(Q, elem) == true)

  then elem == deq(Q) too.

  Irrespective of how the ADT works from the inside (and of course assuming no intervening dequeue)

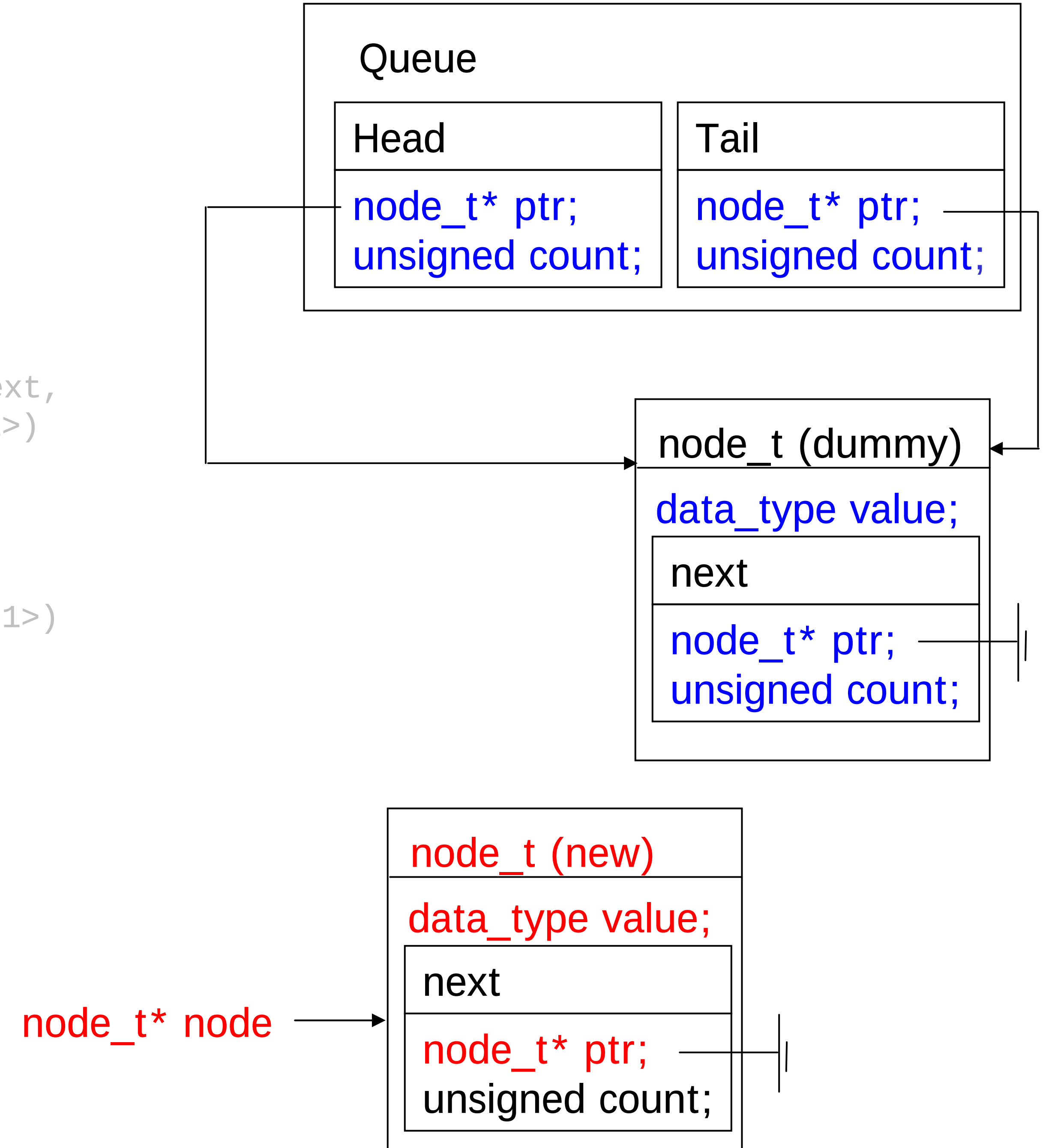
Blocking vs. Non-Blocking

- Non-Blocking requires CAS or LL/SC (or their variants)
- Blocking requires special care and interacts with the scheduler
- Blocking incurs possibly unpredictably long latencies
- In blocking algorithms, deadlocks may happen

Algorithm Walkthrough

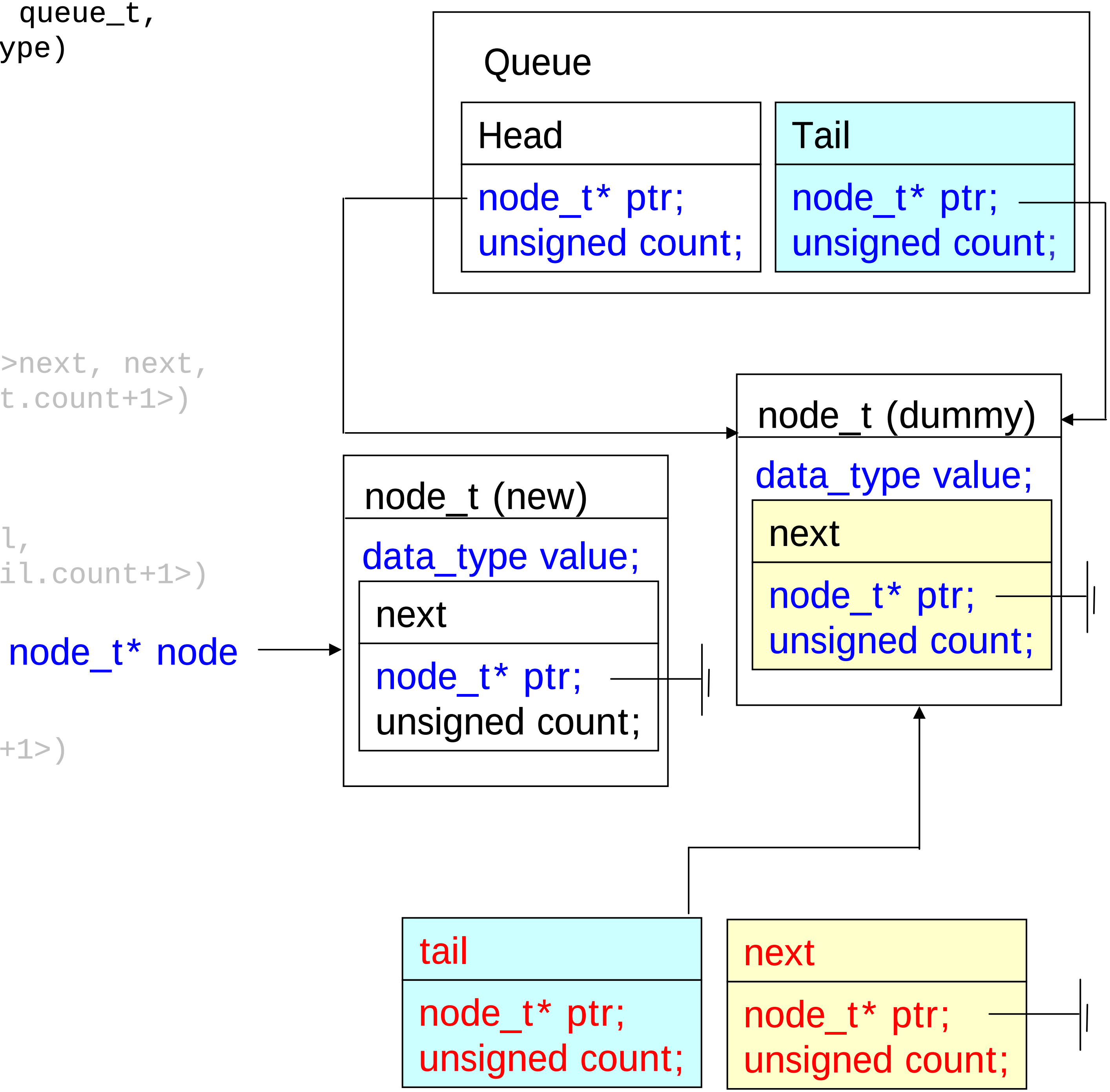
- Implementation decisions/ideas:
  - The list is always connected
  - Nodes are inserted only at the end
  - Nodes are deleted only from the beginning
  - “Head” points to the first node.
  - “Tail” points to a node in the list
  - There is always at least one node in the list
  - List is initialized with a dummy node

```
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>)
```

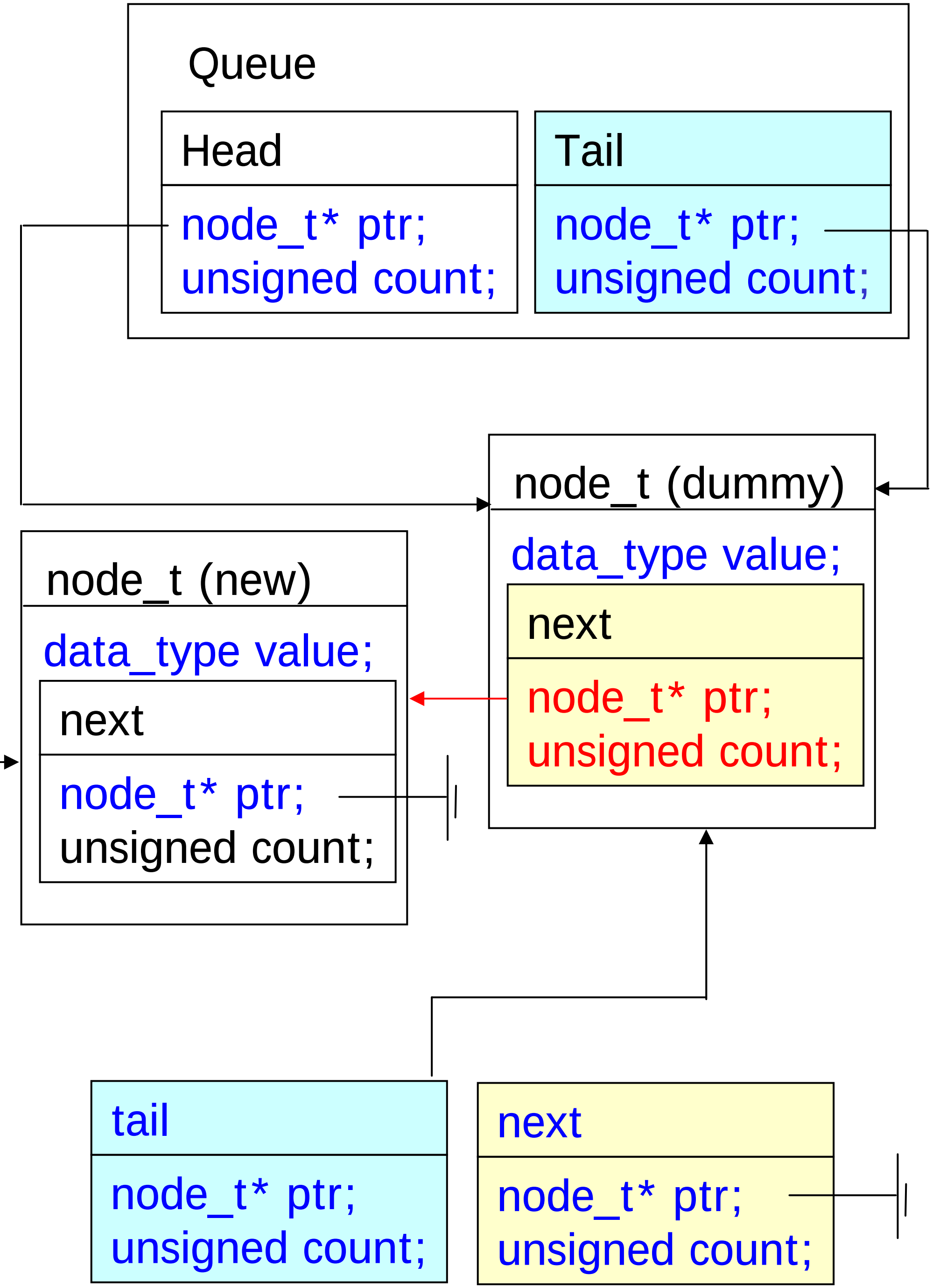


Legends:
Initialized
Changed in this step

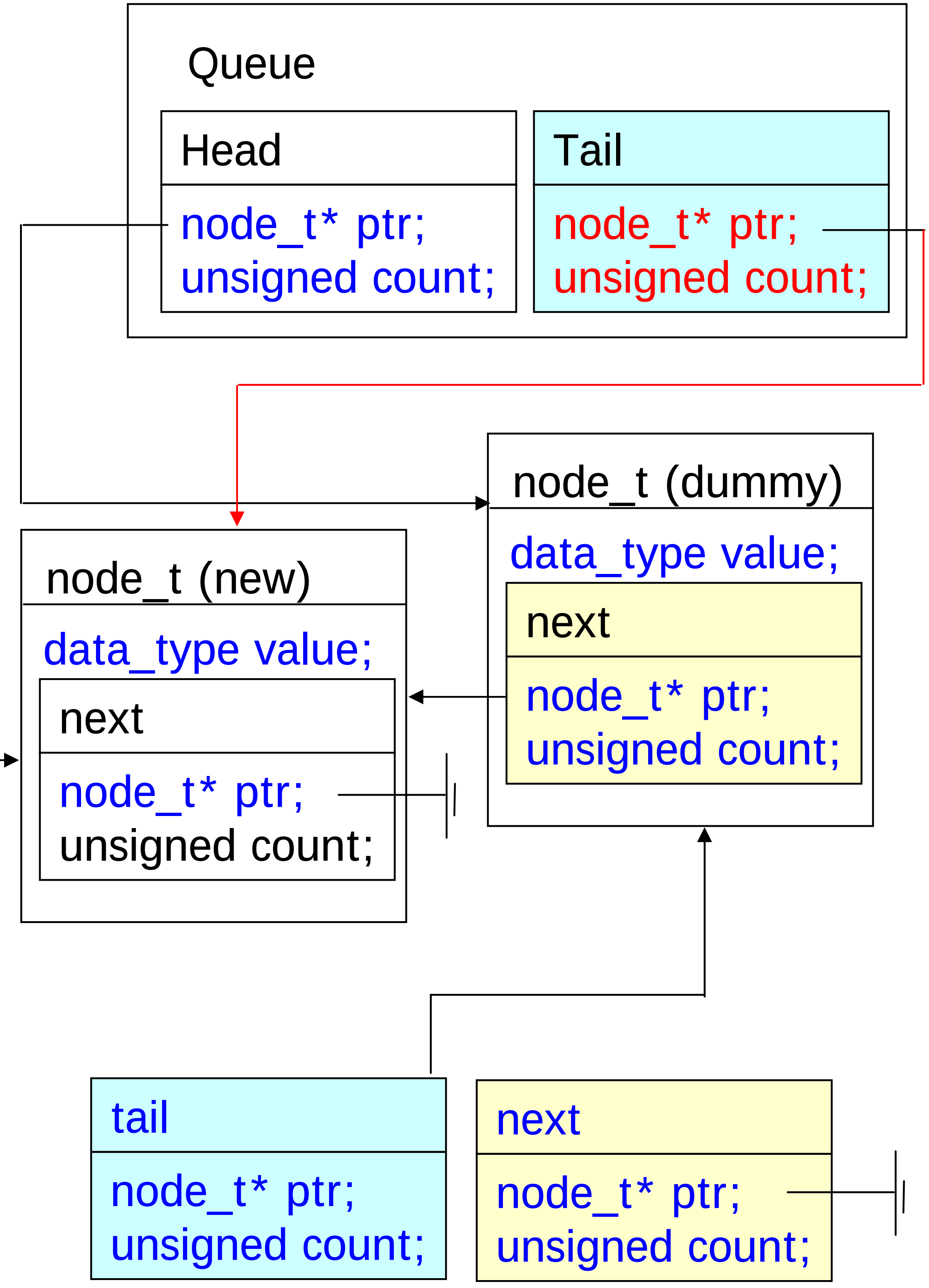
```
enqueue(Q: pointer to queue_t,
        value: data_type)
node = new node()
node->value = value
node->next.ptr = NULL
loop
tail = Q->Tail
next = tail.ptr->next
if tail == Q->Tail
  if next.ptr == NULL
    if CAS(&tail.ptr->next, next,
           <node, next.count+1>)
      break
    endif
  else
    CAS(&Q->Tail, tail,
        <next.ptr, tail.count+1>)
  endif
endif
endloop
CAS(&Q->Tail, tail,
    <node, tail.count+1>)
```



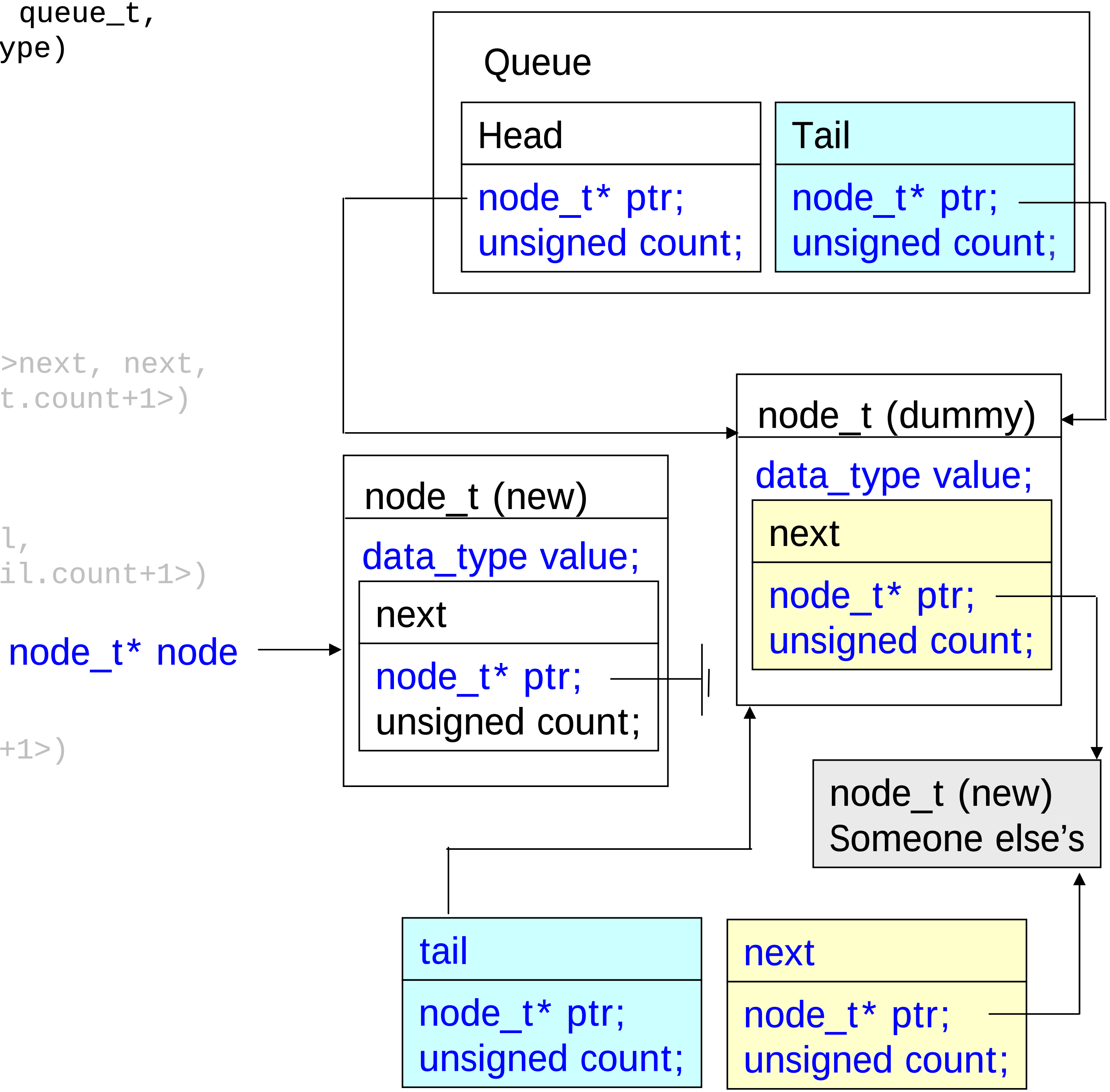
```
enqueue(Q: pointer to queue_t,
        value: data_type)
node = new node()
node->value = value
node->next.ptr = NULL
loop
tail = Q->Tail
next = tail.ptr->next
if tail == Q->Tail
  if next.ptr == NULL
    if CAS(&tail.ptr->next, next,
          <node, next.count+1>)
      break
    endif
  else
    CAS(&Q->Tail, tail,
        <next.ptr, tail.count+1>)
  endif
endif
endloop
CAS(&Q->Tail, tail,
    <node, tail.count+1>)
```



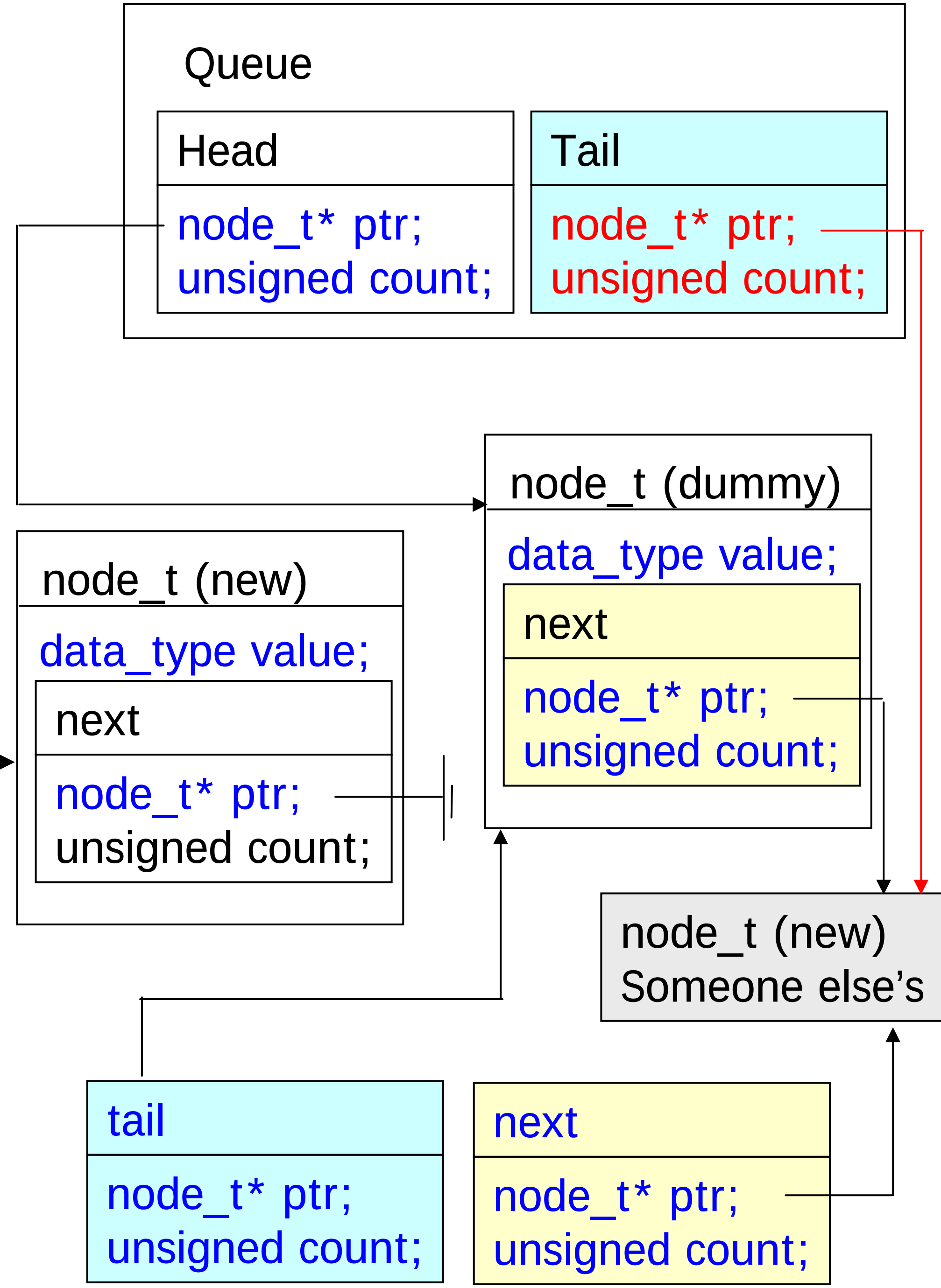
```
enqueue(Q: pointer to queue_t,
        value: data_type)
node = new node()
node->value = value
node->next.ptr = NULL
loop
tail = Q->Tail
next = tail.ptr->next
if tail == Q->Tail
  if next.ptr == NULL
    if CAS(&tail.ptr->next, next,
           <node, next.count+1>)
      break
    endif
  else
    CAS(&Q->Tail, tail,
        <next.ptr, tail.count+1>)
  endif
endif
endloop
CAS(&Q->Tail, tail,
    <node, tail.count+1>)
```



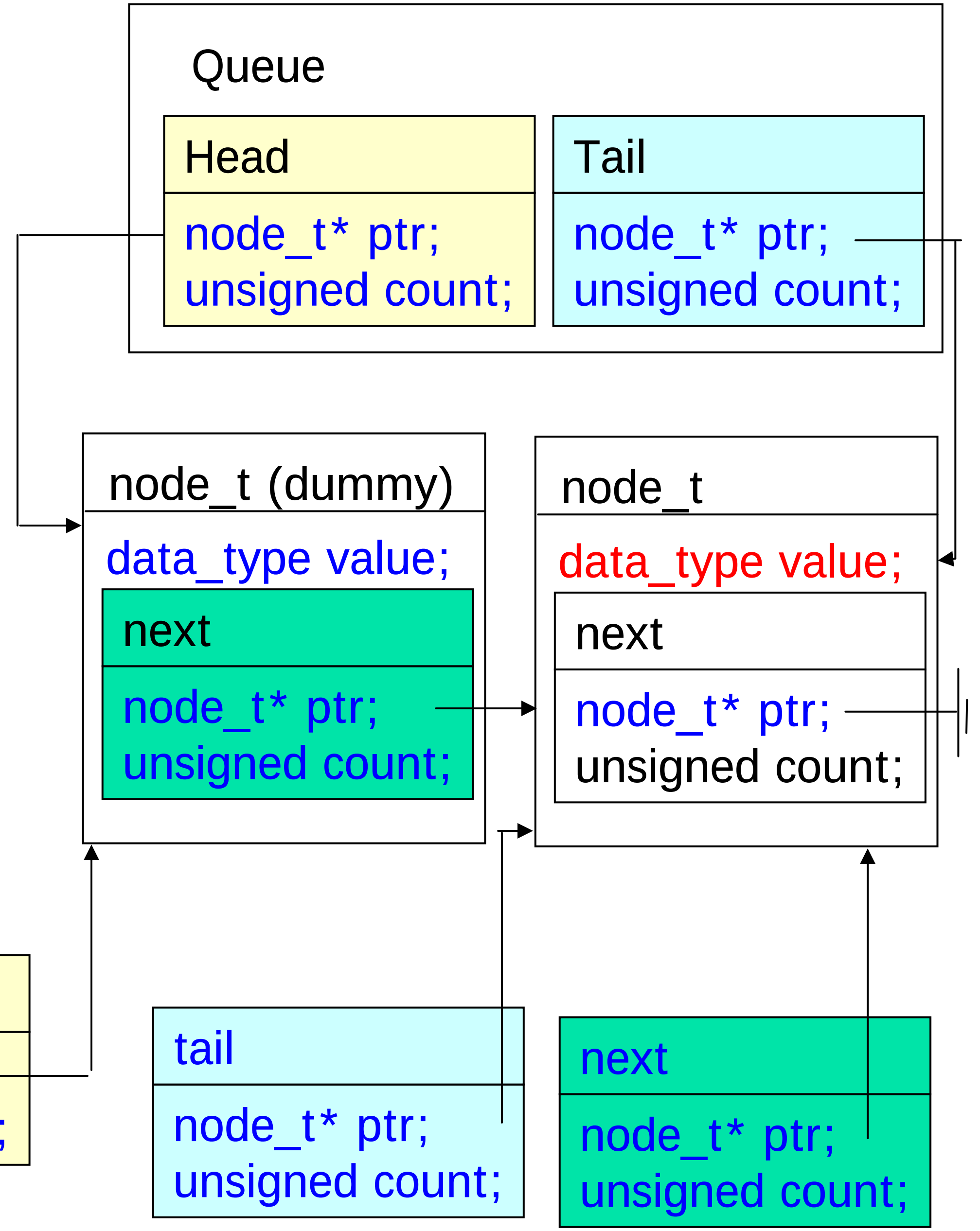
```
enqueue(Q: pointer to queue_t,
        value: data_type)
node = new node()
node->value = value
node->next.ptr = NULL
loop
tail = Q->Tail
next = tail.ptr->next
if tail == Q->Tail
  if next.ptr == NULL
    if CAS(&tail.ptr->next, next,
           <node, next.count+1>)
      break
    endif
  else
    CAS(&Q->Tail, tail,
        <next.ptr, tail.count+1>)
  endif
endif
endloop
CAS(&Q->Tail, tail,
    <node, tail.count+1>)
```

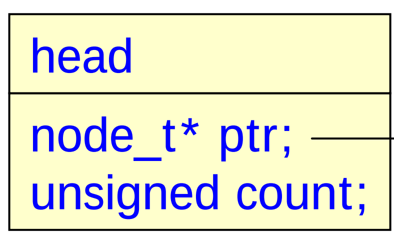


```
enqueue(Q: pointer to queue_t,
        value: data_type)
node = new node()
node->value = value
node->next.ptr = NULL
loop
tail = Q->Tail
next = tail.ptr->next
if tail == Q->Tail
  if next.ptr == NULL
    if CAS(&tail.ptr->next, next,
           <node, next.count+1>)
      break
    endif
  else
    CAS(&Q->Tail, tail,
        <next.ptr, tail.count+1>)
  endif
endif
endloop
CAS(&Q->Tail, tail,
    <node, tail.count+1>)
```

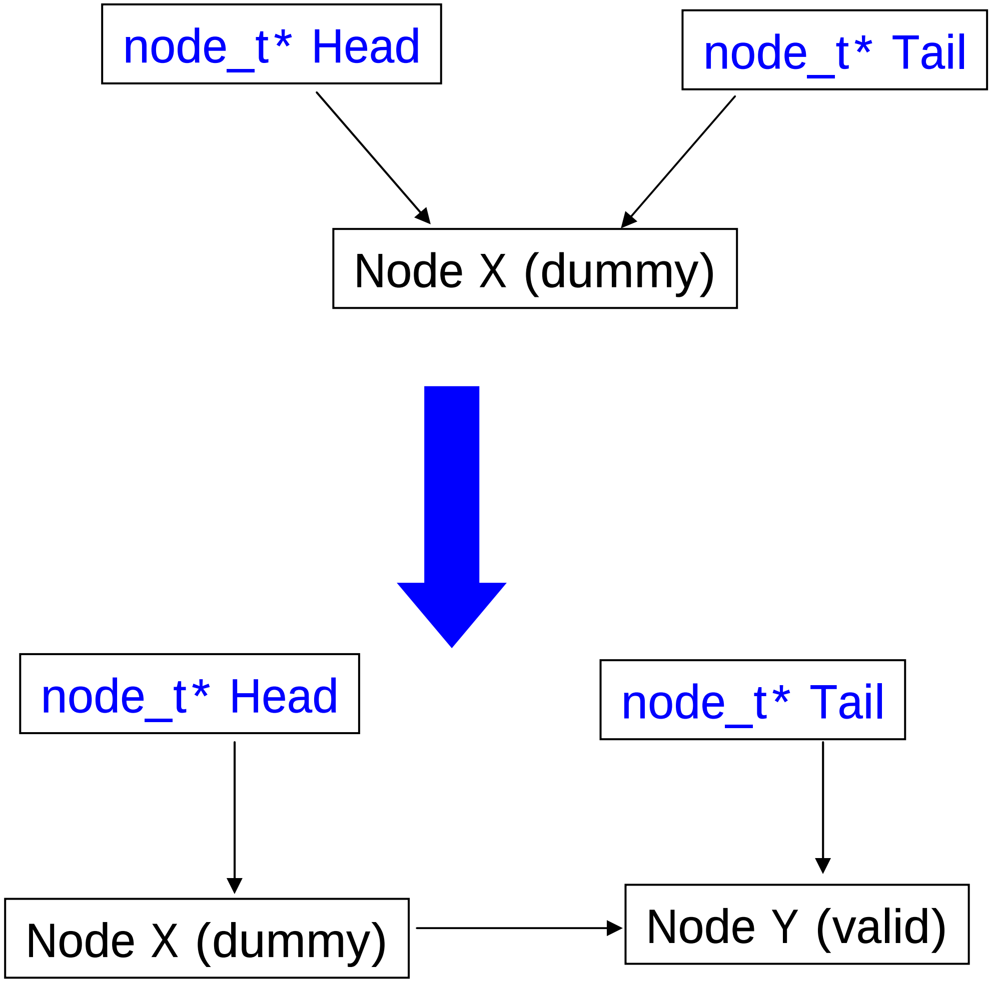


```
dequeue(Q: pointer to queue_t,
        pvalue: pointer to data_type
        ): boolean
loop
head = Q->Head
tail = Q->Tail
next = head->next (Head.ptr->next)
if head == Q->Head
  if head.ptr == tail.ptr
    if next.ptr == NULL
      return FALSE
    endif
    CAS(&Q->Tail, tail,
        <next.ptr, 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
```

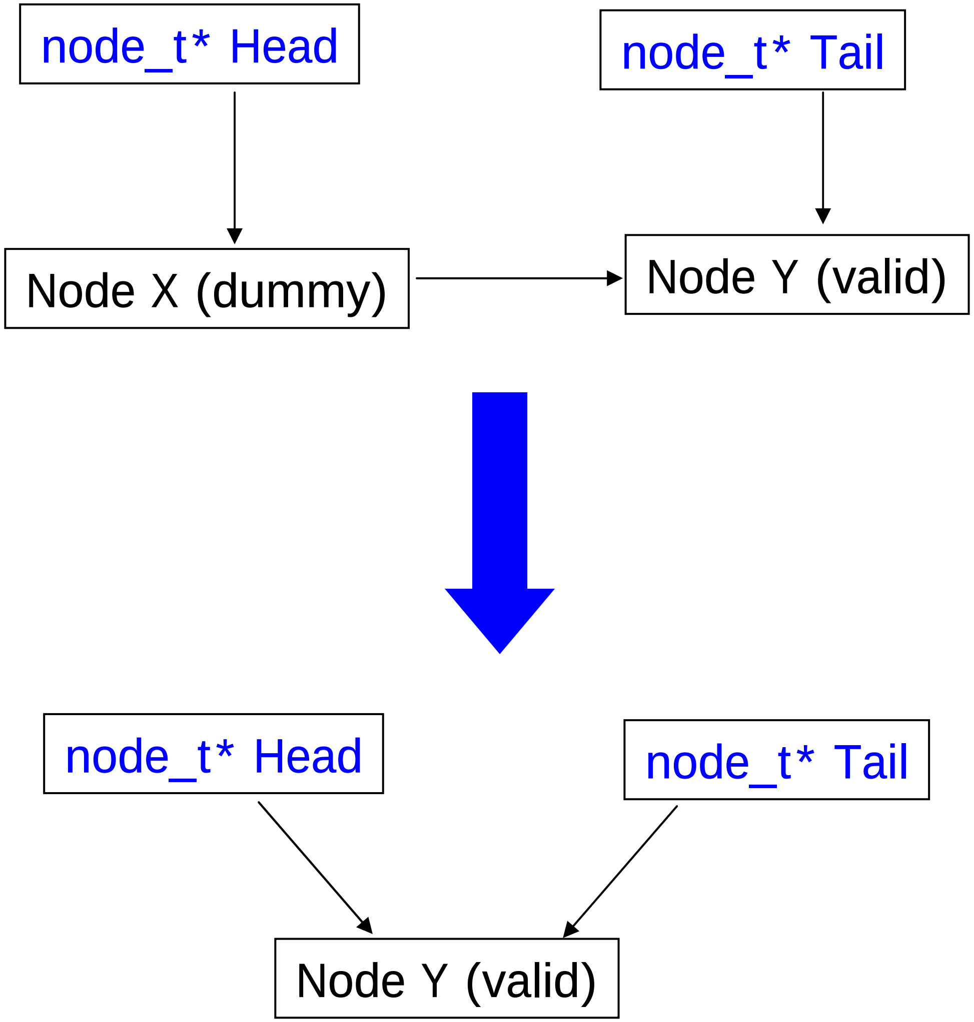




```
enqueue(Q: pointer to queue_t,
          value: data_type)
node = new node()
node–>value = value
node–>next.ptr = NULL
lock(&Q–>T lock)
          Q–>Tail–>next = node
          Q–>Tail = node
unlock(&Q–>T lock)
```



```
dequeue(Q: pointer to queue_t,
          pvalue: pointer to data_type
          ): boolean
lock(&Q–>H lock)
          node = Q–>Head
          new_head = node–>next
          If new_head == NULL
                    unlock(&Q–>H lock)
                    return FALSE
          endif
          *pvalue = new_head–>value
          Q–>Head = new_head
unlock(&Q–>H lock)
free(node)
return TRUE
```



Note: Returns the value of Y,
but frees the first node!

```
enqueue(Q: pointer to queue_t,
        value: data_type)
node = new node()
node–>value = value
node–>next.ptr = NULL
lock(&Q–>T lock)
        Q–>Tail–>next = node
        Q–>Tail = node
unlock(&Q–>T lock)
```

Fictitious synchronization point,
Allows an enqueuer and a dequeuer to
execute simultaneously

```
dequeue(Q: pointer to queue_t,
        pvalue: pointer to data_type
        ): boolean
lock(&Q–>H lock)
        node = Q–>Head
        new_head = node–>next
        If new_head == NULL
                unlock(&Q–>H lock)
                return FALSE
        endif
        *pvalue = new_head–>value
        Q–>Head = new_head
unlock(&Q–>H lock)
free(node)
return TRUE
```

Comparison to previous algorithms

- Similarities
  - Optimistic concurrency
  - Reliance on Compare-and-Swap
  - The “count” field is the “version” in previous papers
- Differences
  - No delayed reclamation of memory
  - No use of DCAS but a more expensive one (compare the pointer and the count)

Performance (1)

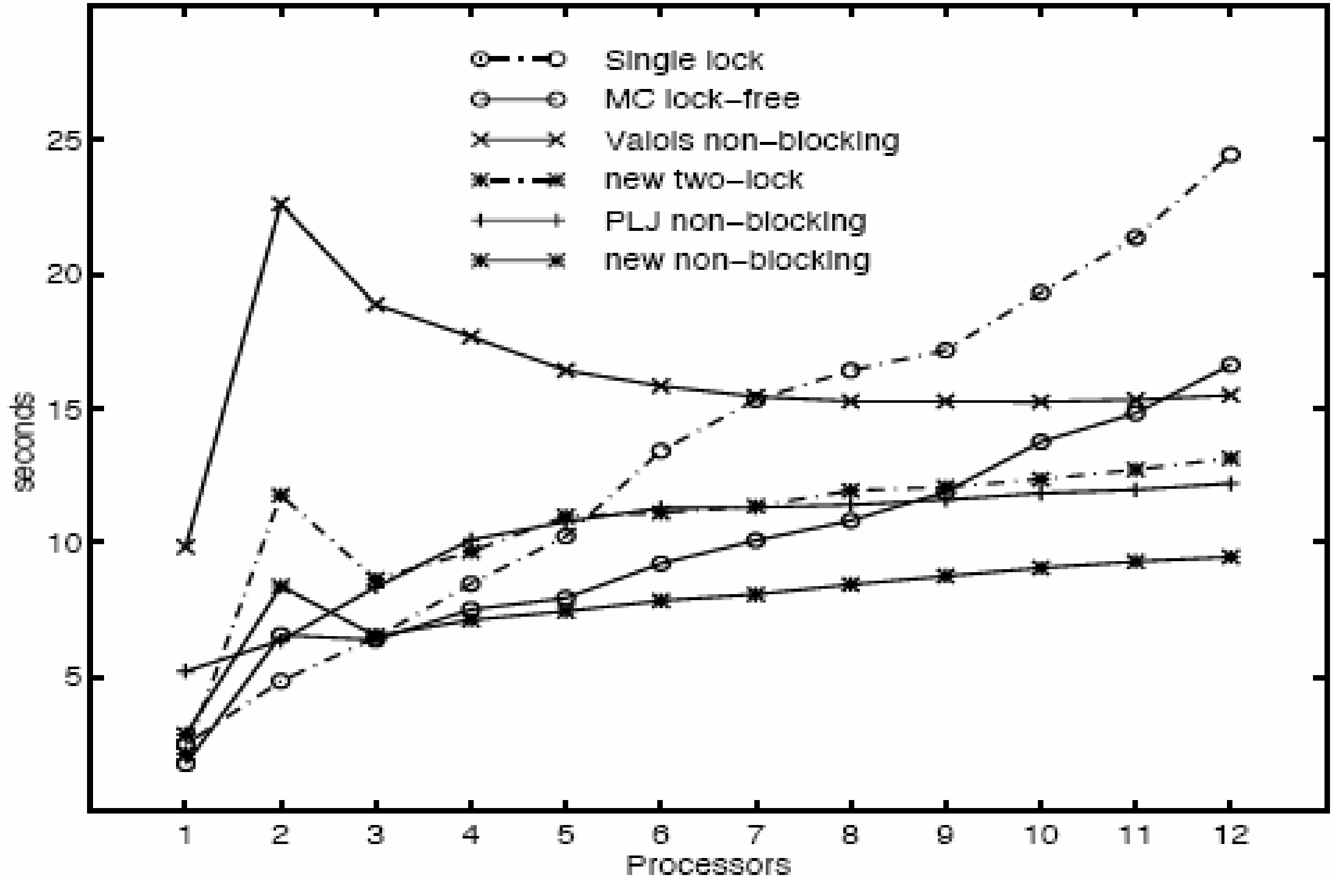


Figure 3: Net execution time for one million enqueue/dequeue pairs on a dedicated multiprocessor.

Performance (2)

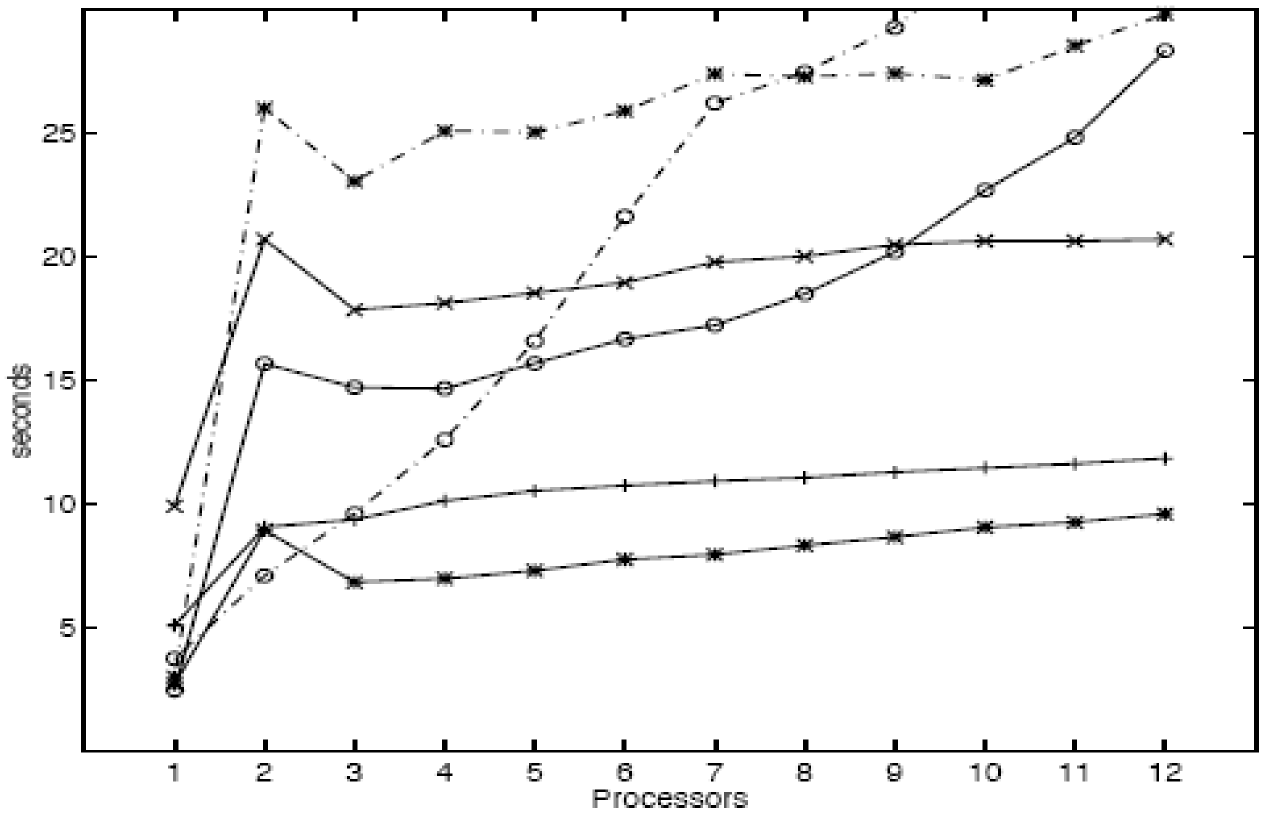


Figure 4: Net execution time for one million enqueue/dequeue pairs on a multiprogrammed system with 2 processes per processor.

Performance (3)

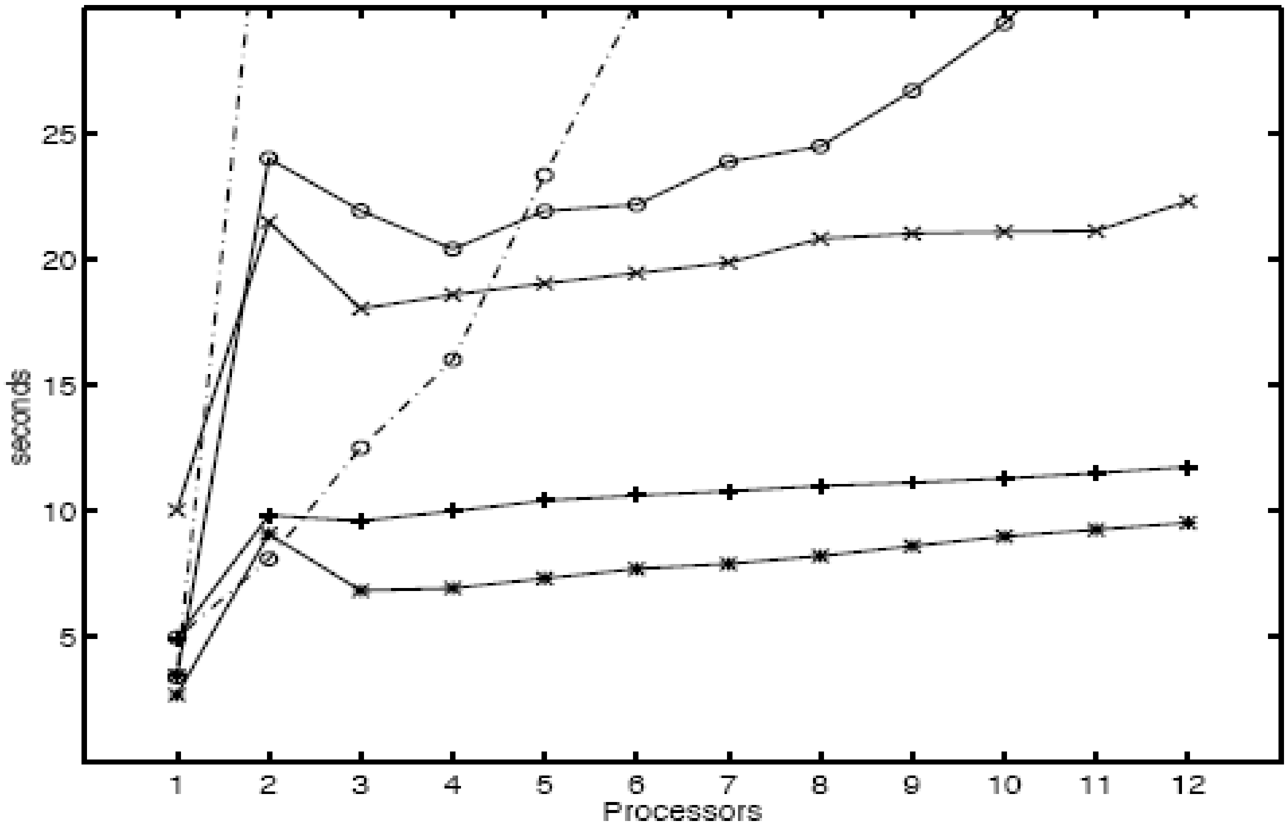


Figure 5: Net execution time for one million enqueue/dequeue pairs on a multiprogrammed system with 3 processes per processor.