Non-Blocking Algorithms

Abdul Moid Munawar, William Leyman, and Jacob Lovell

1

Introduction

2

Blocking Algorithms

- One thread relies on an action by another thread to continue its execution
- Includes all algorithms that may wait to acquire a lock

3

Non-Blocking Algorithms

- there is never a reachable state of the system in which some thread is unable to make forward progress
- “failure or suspension of any thread cannot cause failure or suspension of another thread” (Wikipedia)

4

Three kinds of Non-blocking algorithms

- Obstruction Free
- Lock Free
- Wait Free

5

Obstruction Free

Guarantees that a thread running in isolation will make progress

6

Lock Free

Guarantees that some process will complete its operation in a finite amount of time, even if other processes halt

7

Wait Free

Operations can guarantee that EVERY non-faulting process will complete its operation in a finite amount of time

8

Linearizability

- Property of high level concurrent objects
- Operations appear to occur instantaneously
- Each operation should have one linearization point

9

Lock-Based Counter

```
c = 0
Lock L

void inc():
       acquire(L)
       c++
       release(L)

int val():
       return c
```


10

Obstruction Free Counter

```
c = 0

void inc():
       do:
       tmp = c
       LL(&c)
    while(SC(&c, tmp++))

int val():
       return c
```


11

Lock-Free Counter

```
c = 0

void inc():
    tmp = c
    while(!CAS(&c, tmp, tmp+1)):
              tmp = c

int val():
       return c
```


12

Wait-free Counter

```
// each i represents a thread id
C[i] = 0 for all i

void inc():
   C[self]++

int val():
   rtn := 0
   for i in [1..N]
      rtn +:= C[i]
      return rtn
```


A Solution to the ABA Problem

- Add a counter to each word
- Even if the word is the same, the counter will be different

Memory Consistency Syntax

- fence(RW||RW)
- load(RW||RW)
- store(value, RW||RW)

Memory Consistency Example

```
// initially x = f = 0
Thread 1:
       x := foo()
       f := 1
```

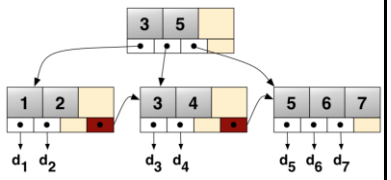
```
Thread 2:
       while f = 0:
              // spin
       y := 1/x
```

Lock-Free B+ Tree

17

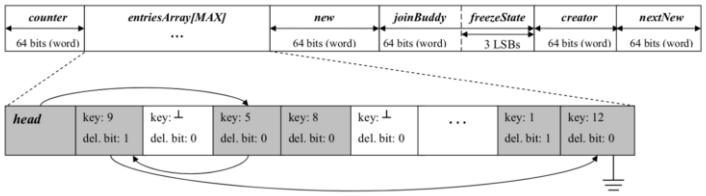
What is a B+ Tree?

- A Binary Tree of only keys
- Used for File systems
- The amount of data allowed per node has an upper and lower bound



18

Chunks



19

Basic Operations/Balancing

- Insert
- Delete
- Search
- Split
- Join
- Copy

20

Algorithm 3 (a) Search, Insert, and Delete – High Level Methods.

(a) BOOL SearchInBtree (key, *data) {

```
1: Node* node = FindLeaf(key);
2: return SearchInChunk(&(node→chunk), key, data);
}
```

(b) BOOL InsertToBtree (key, data) {

```
1: Node* node = FindLeaf(key);
2: if (node→freezeState == INFANT) helpInfant(node);        // Help infant node
3: return InsertToChunk(&(node→chunk), key, data);
}
```

(c) BOOL DeleteFromBtree (key, data) {

```
1: Node* node = FindLeaf(key);
2: if (node→freezeState == INFANT) helpInfant(node);        // Help infant node
3: return DeleteInChunk(&(node→chunk), key);
}
```

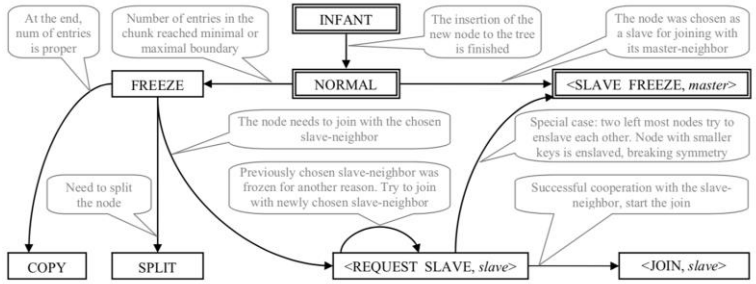



Figure 2: The state transitions of the freeze state of a node. The initial states are presented in the boxes with the double border.

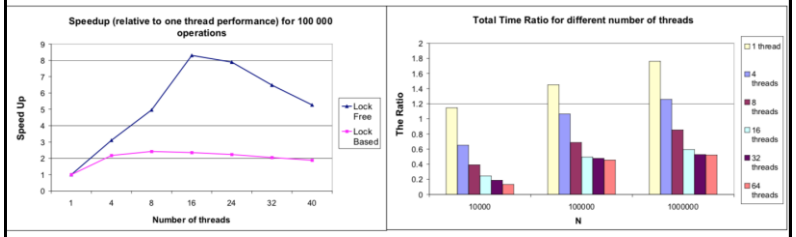
How is Linearizability Achieved?

- Node is only modified after replacement
- Special care in selecting join node, where both will share the same parent

How is Lock freedom achieved?

- CAS is used
- Freeze State and Infant State
- Limited Join Selection

Performance



25

Non Blocking Queue

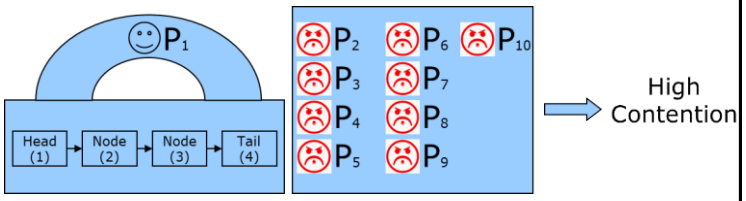
Algorithm Creators: Maged M. Michael and Michael L. Scott

Presentor: Abdul Moid Munawar

26

Single Lock based Concurrent Queue

- Single Lock Queue locks entire queue

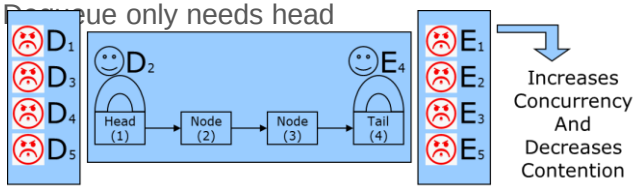


Credits: Hussain Tinwala

27

Two Lock based Concurrent Queue

- Only locks head or tail node
- Enqueue only needs tail
- Dequeue only needs head



Credits: Hussain Tinwala

28

Michael and Scott Queue

- Lock-free but not wait-free
- Prevents Livelock
- Starvation is possible

29

Queue setup

```
type ptr = ⟨node* p, int c⟩                    // counted pointer
type node
    value val
    ptr next
class queue
    ptr head
    ptr tail
void queue.init()
    node* n := new node(⊥, null)               // initial dummy node
    head.p := tail.p := n
```


30

Queue Operations

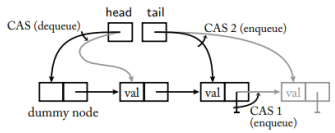


Figure 8.3: Operation of the M&S queue. After appropriate preparation ("snapshotting"), dequeue reads a value from the second node in the list, and updates head with a single CAS to remove the old dummy node. In enqueue, two CASes are required: one to update the next pointer in the previous final node; the other to update tail.

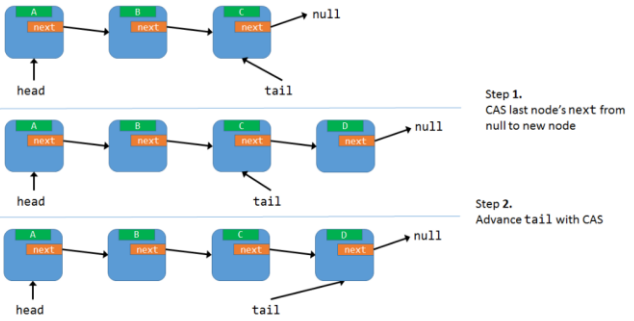
31

Enqueue Detail

```
void queue.enqueue(value v):
    node* w := new node(v, null);  fence(W||W)     // allocate node for new value
    ptr t, n
    loop
        t := tail.load()                           // counted pointers
        n := t.p→next.load()
        if t = tail.load()                         // are t and n consistent?
            if n.p = null                          // was tail pointing to the last node?
                if CAS(&t.p→next, n, ⟨w, n.c+1⟩)   // try to add w at end of list
                    break                          // success; exit loop
            else                                   // tail was not pointing to the last node
                (void) CAS(&tail, t, ⟨n.p, t.c+1⟩) // try to swing tail to next node
    (void) CAS(&tail, t, ⟨w, t.c+1⟩)               // try to swing tail to inserted node
```


32

Enqueue Detail

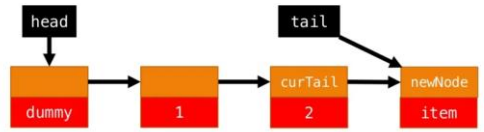


33

Enqueue Possibilities

```
void queue.enqueue(value v):
    node* w := new node(v, null);  fence(W||W)      // allocate node for new value
    ptr t, n
    loop
        t := tail.load()                            // counted pointers
        n := t.p→next.load()
        if t = tail.load()                          // are t and n consistent?
            if n.p = null                           // was tail pointing to the last node?
                if CAS(&t.p→next, n, ⟨w, n.c+1⟩)    // try to add w at end of list
                    break                           // success; exit loop
            else                                    // tail was not pointing to the last node
                (void) CAS(&tail, t, ⟨n.p, t.c+1⟩)  // try to swing tail to next node
    (void) CAS(&tail, t, ⟨w, t.c+1⟩)                // try to swing tail to inserted node
```

True



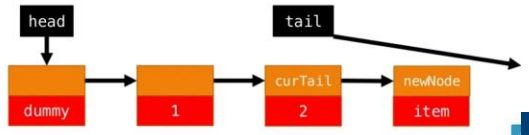
34

Enqueue Possibilities

```
void queue.enqueue(value v):
    node* w := new node(v, null);  fence(W||W)      // allocate node for new value
    ptr t, n
    loop
        t := tail.load()                            // counted pointers
        n := t.p→next.load()
        if t = tail.load()                          // are t and n consistent?
            if n.p = null                           // was tail pointing to the last node?
                if CAS(&t.p→next, n, ⟨w, n.c+1⟩)    // try to add w at end of list
                    break                           // success; exit loop
            else                                    // tail was not pointing to the last node
                (void) CAS(&tail, t, ⟨n.p, t.c+1⟩)  // try to swing tail to next node
    (void) CAS(&tail, t, ⟨w, t.c+1⟩)                // try to swing tail to inserted node
```

True

False



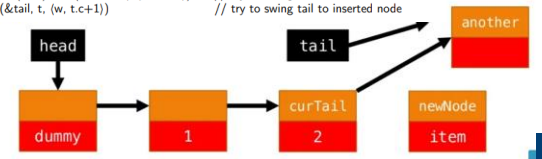
35

Enqueue Possibilities

```
void queue.enqueue(value v):
    node* w := new node(v, null);  fence(W||W)      // allocate node for new value
    ptr t, n
    loop
        t := tail.load()                            // counted pointers
        n := t.p→next.load()
        if t = tail.load()                          // are t and n consistent?
            if n.p = null                           // was tail pointing to the last node?
                if CAS(&t.p→next, n, ⟨w, n.c+1⟩)    // try to add w at end of list
                    break                           // success; exit loop
            else                                    // tail was not pointing to the last node
                (void) CAS(&tail, t, ⟨n.p, t.c+1⟩)  // try to swing tail to next node
    (void) CAS(&tail, t, ⟨w, t.c+1⟩)                // try to swing tail to inserted node
```

False

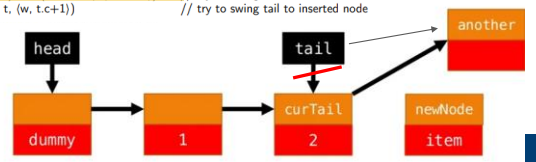
True



36

Enqueue Possibilities

```
void queue.enqueue(value v):
    node* w := new node(v, null);  fence(W||W)      // allocate node for new value
    ptr t, n
    loop
        t := tail.load()                            // counted pointers
        n := t.p→next.load()
        if t = tail.load()                          // are t and n consistent?
            if n.p = null          False            // was tail pointing to the last node?
                if CAS(&t.p→next, n, ⟨w, n.c+1⟩)    // try to add w at end of list
                    break                           // success; exit loop
            else                                    // tail was not pointing to the last node
                (void) CAS(&tail, t, ⟨n.p, t.c+1⟩)  // try to swing tail to next node  True
    (void) CAS(&tail, t, ⟨w, t.c+1⟩)                // try to swing tail to inserted node
```

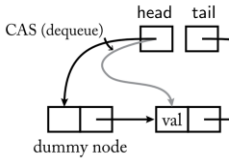


Dequeue Detail

```
value queue.dequeue():
    ptr h, t, n
    loop
        h := head.load()                            // counted pointers
        t := tail.load()
        n := h.p→next.load()
        value rtn
        if h = head.load()                          // are h, t, and n consistent?
            if h.p = t.p                            // is queue empty or tail falling behind?
                if n.p = null return ⊥              // empty; return failure
                (void) CAS(&tail, t, ⟨n.p, t.c+1⟩)  // tail is falling behind; try to update
            else                                    // no need to deal with tail
                // read value before CAS; otherwise another dequeue might free n
                rtn := n.p→val.load()
                if CAS(&head, h, ⟨n.p, h.c+1⟩)      // try to swing head to next node
                    break                           // success; exit loop
    fence(W||W)                                     // link node out before deleting!
    free_for_reuse(h.p)                             // type-preserving
    return rtn                                      // queue was nonempty; return success
```


Dequeue Detail

```
value queue.dequeue():
    ptr h, t, n
    loop
        h := head.load()                            // counted pointers
        t := tail.load()
        n := h.p→next.load()
        value rtn
        if h = head.load()   Make sure head hasn't changed // are h, t, and n consistent?
            if h.p = t.p                            // is queue empty or tail falling behind?
                if n.p = null return ⊥              // empty; return failure
                (void) CAS(&tail, t, ⟨n.p, t.c+1⟩)  // tail is falling behind; try to update
            else                                    // no need to deal with tail
                // read value before CAS; otherwise another dequeue might free n
                rtn := n.p→val.load()
                if CAS(&head, h, ⟨n.p, h.c+1⟩)      // try to swing head to next node
                    break                           // success; exit loop
    fence(W||W)                                     // link node out before deleting!
    free_for_reuse(h.p)                             // type-preserving
    return rtn                                      // queue was nonempty; return success
```



Performance

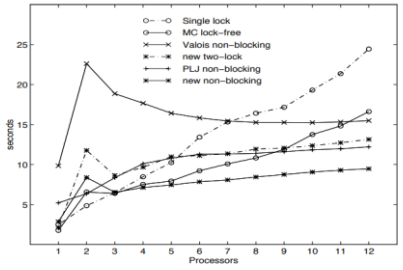


Figure 3: Net execution time for one million enqueue/dequeue pairs on a dedicated multiprocessor.

Performance

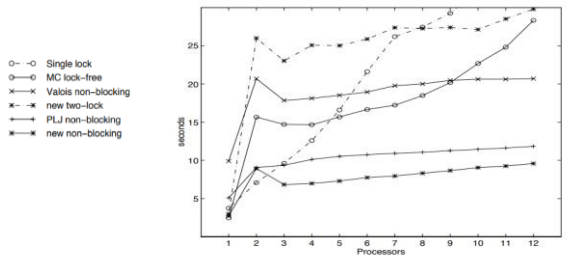


Figure 4: Net execution time for one million enqueue/dequeue pairs on a multiprogrammed system with 2 processes per processor.

41

Performance

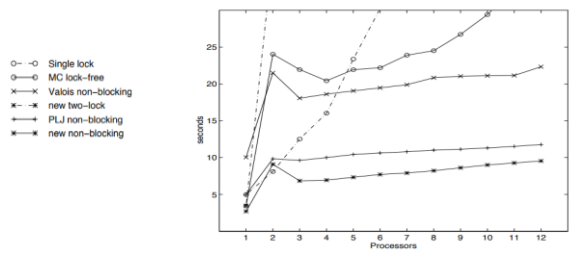


Figure 5: Net execution time for one million enqueue/dequeue pairs on a multiprogrammed system with 3 processes per processor.

42

Performance Conclusions

- In all three graphs, the new non-blocking queue outperforms all of the other alternatives when three or more processors are active
- The two-lock algorithm outperforms the one-lock algorithm when more than 5 processors are active on a dedicated system

43

References

http://concurrencyfreaks.blogspot.com/2017/01/doublelink-low-overhead-lock-free-queue.html

https://wiki.eecs.yorku.ca/course_archive/2007-08/F/6490A/_media/presentations:hussain.ppt

Alexey Fyodorov Slides: https://www.slideshare.net/23derevo/nonblocking-michaelscott-queue-algorithm?from_action=save

https://www.cs.rochester.edu/u/scott/papers/1996_PODC_queues.pdf

44