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

Maged M.Michael
Michael L.Scott
Department of Computer Science
Univeristy of Rochester

Presented by: Jun Miao

Outline

- Introduction
- Related work
- 2-lock FIFO queue algorithm
- Non-blocking FIFO queue algorithm
- Experiment
- Conclusion
- Advantages and Disadvantages

Introduction

- FIFO queue: A linked list of items with two interface methods, namely, enqueue and dequeue.

- Problem: How to make enqueues and dequeues work concurrently and correctly on a shared FIFO queue

Some Definitions

- Wait-free
  - All threads or processes can make progress in finite steps
- Non-blocking
  - Some threads or processes are guaranteed to make progress in finite steps
- Obstruction-free
  - One thread or process is guaranteed to make progress in finite steps
- Lock-free
  - No deadlock or Not using locks?

Methods

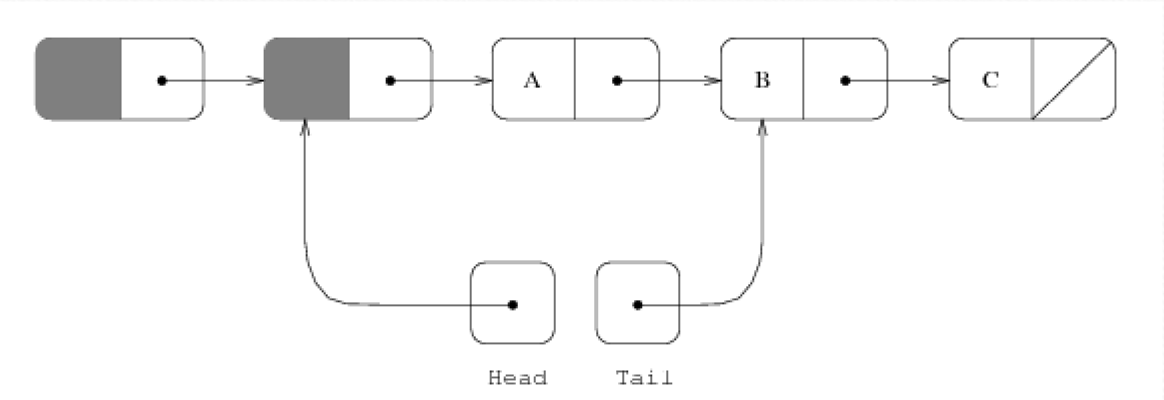
- Blocking: utilizes locks to prevent conflicts between competing processes and threads
  - Most significant feature: Slow processes and threads may block fast ones

- Non-Blocking: ensures that threads competing for a shared resource do not have their execution indefinitely postponed by mutex or semaphore.

Related Work

- Hwang and Briggs[1984], Sites[1980] and Stones[1990] present lock-free algorithms based on compare_and_swap.
- Gottlieb et al.[1983] and Mellor-Crummey[1987] present algorithms that are lock-free but not non-blocking.
- Prakash, Lee and Johnson[1991,1994] present a non-blocking algorithm by taking snapshots of the queue
- Valois[1994,1995] presents a list-based non-blocking algorithm.

The FIFO queue

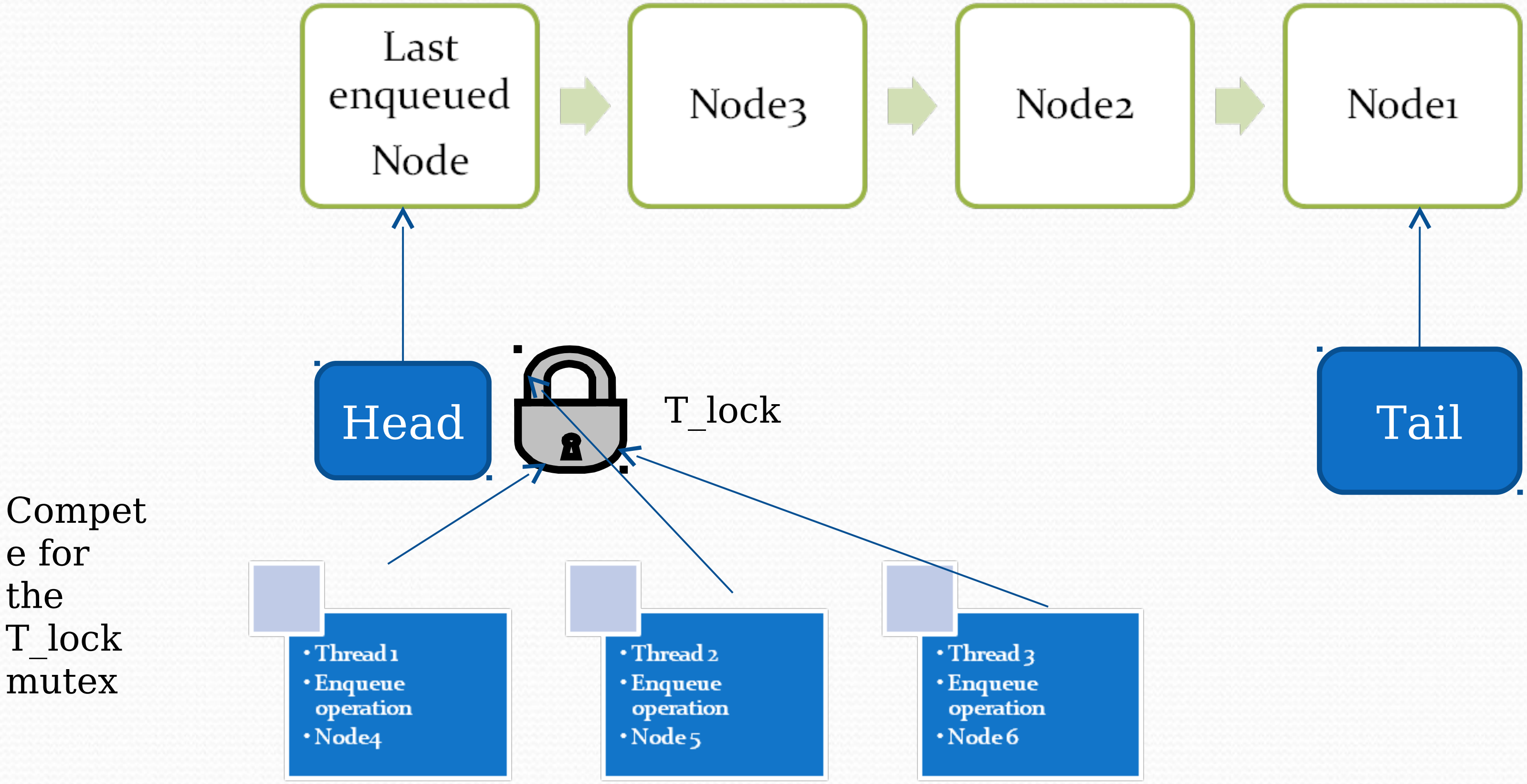


Head: always point to the last dequeued point
Tail: Should point to the last node in the queue

The only inconsistent condition: a new node is enqueued as the last node, but Tail is not updated to point to it.

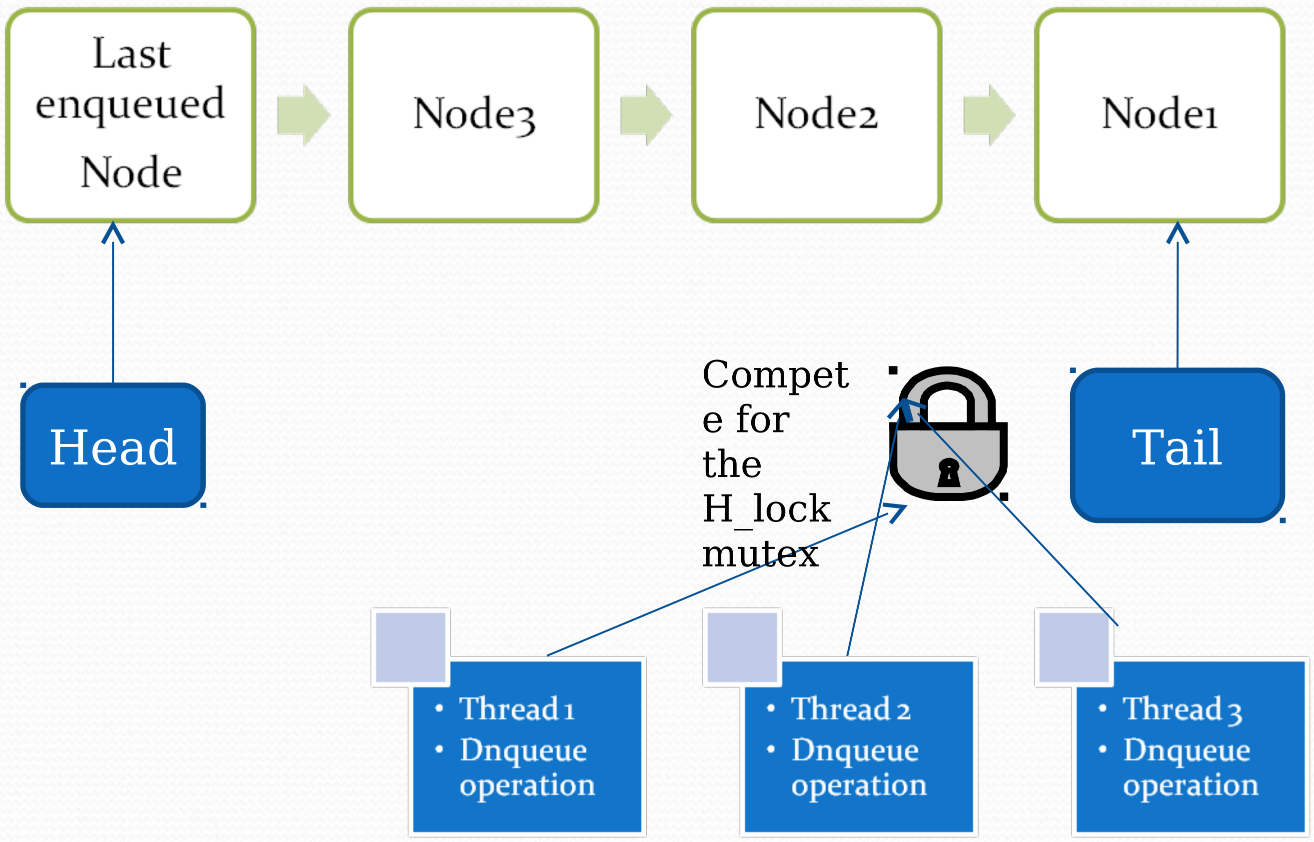
2-Lock concurrent FIFO queue algorithm

- Enqueue operation



2-Lock concurrent FIFO queue algorithm(Cont'd)

- **Dequeue operation**



2-Lock concurrent FIFO queue algorithm (Cont'd)

```
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
```

Compare And Swap

- A special instruction that atomically compares the contents of a memory location to a given value and, if they are the same, modifies the contents of that memory location to a given new value.

C code of CAS:

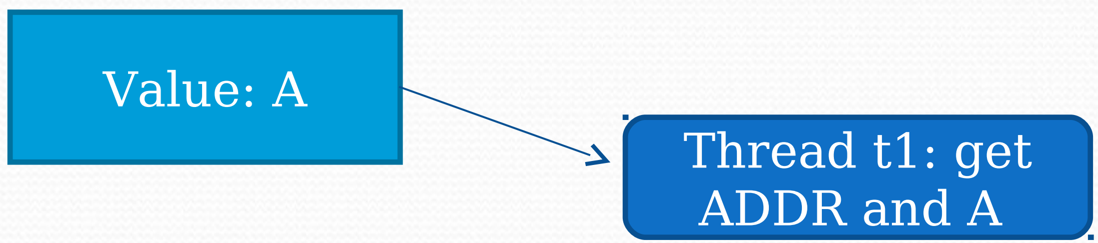
```
int compare_and_swap (int *word, int testval, int newval)
 {
   int oldval; oldval = *word;
   if (oldval == testval) *word = newval;
   return oldval;
}
```

ABA Problem

- ABA problem is the enemy of non-blocking algorithm using CAS primitive.

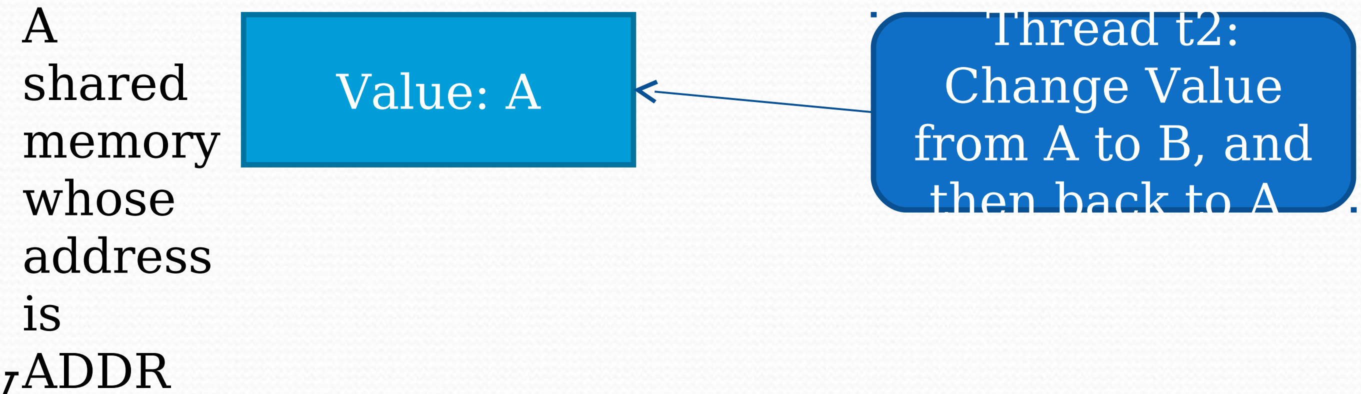
Initially

A shared memory whose address is ADDR

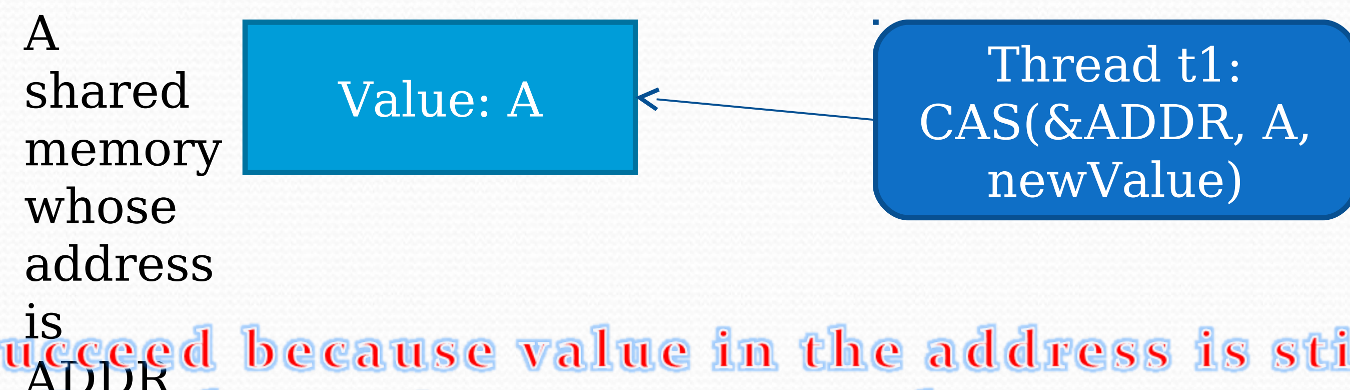


ABA Problem(Cont’d)

- Next



- Finally



CAS will succeed because value in the address is still A. But the environment may change!

Solution

- Associate a counter with each block of memory

<node, counter>

If the node changes, counter++

- Compare not only the value in the address, but also the counters when using CAS

Non-blocking FIFO 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 = Q->Tail = 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?
E8:           if next.ptr == NULL           # Was Tail pointing to the last node?
E9:              if CAS(&tail.ptr->next, next, <node, next.count+1>)   # Try to link node at the end of the linked list
E10:                break                   # Enqueue is done. Exit loop
E11:             endif
E12:          else                          # Tail was not pointing to the last node
E13:             CAS(&Q->Tail, tail, <next.ptr, tail.count+1>)   # Try to swing Tail to the next node
E14:          endif
E15:       endif
E16:    endloop
E17:    CAS(&Q->Tail, tail, <node, tail.count+1>)   # Enqueue is done. Try to swing Tail to the inserted node
```

Non-blocking FIFO queue algorithm(Cont'd)

```
dequeue(Q: pointer to queue_t, pvalue: pointer to data type): boolean
    loop                                                  # Keep trying until Dequeue is done
        head = Q–>Head                                    # Read Head
        tail = Q–>Tail                                    # Read Tail
        next = head–>next                                 # Read Head.ptr–>next
        if head == Q–>Head                                # Are head, tail, and next consistent?
            if head.ptr == tail.ptr                       # Is queue empty or Tail falling behind?
                if next.ptr == NULL                       # Is queue empty?
                    return FALSE                          # Queue is empty, couldn't dequeue
                endif
                CAS(&Q–>Tail, tail, <next.ptr, tail.count+1>)   # Tail is falling behind. Try to advance it
            else                                          # No need to deal with Tail
                # Read value before CAS, otherwise another dequeue might free the next node
                *pvalue = next.ptr–>value
                if CAS(&Q–>Head, head, <next.ptr, head.count+1>)  # Try to swing Head to the next node
                    break                                 # Dequeue is done. Exit loop
                endif
            endif
        endif
    endloop
    free(head.ptr)                                        # It is safe now to free the old dummy node
    return TRUE                                           # Queue was not empty, dequeue succeeded
```

Non-blocking FIFO queue algorithm(Cont'd)

- For a non-blocking algorithm, every CAS primitive will be executed only after step-by-step checks.
- Try to split the procedure into atomic operations to detect every interleaves

Experiment

- Work on 12-processor Silicon Graphics Challenge multiprocessor
- Associate each processes with each processors
- Totally 100,000 pairs enqueue and dequeue operation.
- 100,000/n on each processor. n is the number of processors being used.

Result

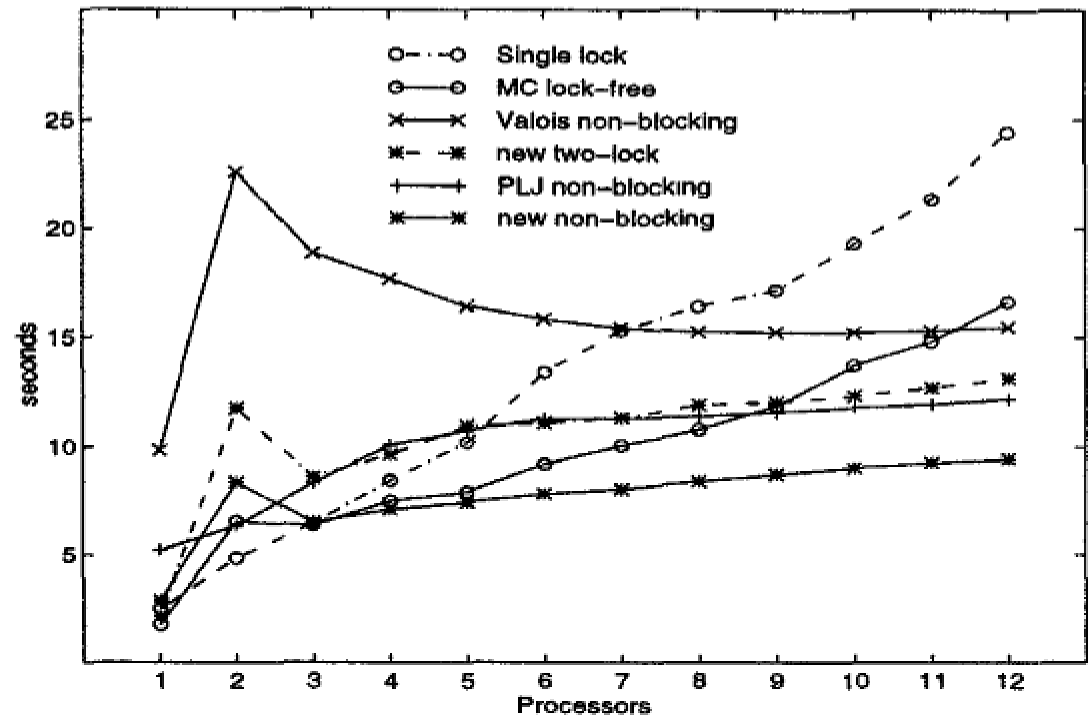


Figure 3: Net execution time for one million enqueue/dequeue pairs on a dedicated multiprocessor.

Conclusion

- The concurrent non-blocking queue algorithm that is simple, practical, and fast. It is the choice of any queue-based application on a multiprocessor with universal atomic primitives.
- The 2-lock is recommended to heavily-utilized queues on machines with simple atomic primitives.

Advantages

- Blocking
  - Easy to implement
  - Can be used universally

- Non-blocking
  - High efficiency
  - Never deadlock
  - Ensures the priority of fast threads

Disadvantages

- Blocking
  - allows slow threads blocking fast threads
  - Deadlock may happen
  - Starvation

- Non-blocking
  - Hard to design
  - depend on what atomic primitives can be provided by systems
  - ABA problem is a potential danger for those who use CAS primitive
  - Cannot completely avoid starvation

Question time

Thank you for your attention!