Lock Free Data Structures

Soham Chakraborty

23.02.2022

Outline

Concurrent objects

- Stack, queue

Safety property

Liveness/progress properties

Concurrent Data Structures & Synchronization

Multiple threads may access the data structure concurrently

Examples:

- Linked list
- Stack
- Queue
- ...

Often referred as concurrent objects (data structure+API methods)

Accessed by a set of methods

- LinkedList: add(), search(), delete()
- Queue: enq(), deq()
- Stack: push(), pop()

Synchronization

Coarse-grained

- Synchronize every access to the object using a global lock
- Example: Lock entire linked-list to add/delete a node

fine-grained

- Partition the object into independent synchronized components
- Example: Lock relevant nodes in a linked-list to add/delete a node

Nonblocking

- No use of lock/unlock
- Use compare_and_swap(CAS) to update atomically

Lock Free/Nonblocking Data Structures

Multiple threads may access the object concurrently

Typically uses compare-and-exchange within a loop

May not be *wait free*

- Every thread completes within a bounded number of steps, regardless of the behavior of other threads

No deadlock, livelock is possible

Node {int data; Node *next; ...}

Queue {Node *head, *tail; ...}

Enqueue:

```
void enq(int x) {
   Node e = new Node(x);
   enqLock.lock();
   tail.next = e;
   tail = e;
   enqLock.unlock();
}
```

Dequeue:

```
int deq() {
   int result;
   deqLock.lock();
   if (head.next == null)
      return ERROR;
   result = head.next.value;
   head = head.next;
   deqLock.unlock();
   return result;
}
```

No deadlock as each tail and head has separate locks

Lock Free Queue: Enqueue

```
void enq(int value) {
    Node node = new Node(value);
    while (true) {
        Node last = tail;
        Node next = last.next;

        if (last == tail) {
            if (next == null) {
                if (CAS(last.next, next, node)) {
                    CAS(tail, last, node);
                    return;
                }
            } else {

                CAS(tail, last, next);
            }
        }
    }
}
```

Lock Free Queue: Enqueue

```
void enq(int value) {
    Node node = new Node(value);  // create a new node
    while (true) {
        Node last = tail;
        Node next = last.next;

        if (last == tail) {
            if (next == null) {
                if (CAS(last.next, next, node)) {
                    CAS(tail, last, node);
                    return;
                }
            } else {

                CAS(tail, last, next);
            }
        }
    }
}
```

Lock Free Queue: Enqueue

```
void enq(int value) {
  Node node = new Node(value);  // create a new node
  while (true) {
    Node last = tail;  // locate the last node
    Node next = last.next;

    if (last == tail) {
      if (next == null) {
        if (CAS(last.next, next, node)) {
          CAS(tail, last, node);
           return;
        }
      } else {

        CAS(tail, last, next);
      }
    }
  }
}
```

Lock Free Queue: Enqueue

```
void enq(int value) {
    Node node = new Node(value);  // create a new node
    while (true) {
        Node last = tail;  // locate the last node
        Node next = last.next;
 // identify the position to append the new node
        if (last == tail) {
            if (next == null) {
                if (CAS(last.next, next, node)) {
                    CAS(tail, last, node);
                     return;
                }
            } else {

                CAS(tail, last, next);
            }
        }
    }
}
```

Lock Free Queue: Enqueue

```
void enq(int value) {
    Node node = new Node(value); // create a new node
    while (true) {
        Node last = tail; // locate the last node
        Node next = last.next;
// identify the position to append the new node
        if (last == tail) {
            if (next == null) { // no successor
                if (CAS(last.next, next, node)) {
                    CAS(tail, last, node);
                    return;
                }
            } else {

                CAS(tail, last, next);
            }
        }
    }
}
```

Lock Free Queue: Enqueue

```
void enq(int value) {
    Node node = new Node(value);  // create a new node
    while (true) {
        Node last = tail;  // locate the last node
        Node next = last.next;
// identify the position to append the new node
        if (last == tail) {
            if (next == null) {  // no successor
                if (CAS(last.next, next, node)) {  // append the new node
                    CAS(tail, last, node);
                    return;
                }
            } else {

                CAS(tail, last, next);
            }
        }
    }
}
```

Lock Free Queue: Enqueue

```
void enq(int value) {
    Node node = new Node(value);  // create a new node
    while (true) {
        Node last = tail;  // locate the last node
        Node next = last.next;
// identify the position to append the new node
        if (last == tail) {
            if (next == null) {  // no successor
                if (CAS(last.next, next, node)) {  // append the new node
                    CAS(tail, last, node);   // new node is the tail
                    return;
                }
            } else {

                CAS(tail, last, next);
            }
        }
    }
}
```

Lock Free Queue: Enqueue

```
1.  void enq(int value) {
2.     Node node = new Node(value);  // create a new node
3.     while (true) {
4.        Node last = tail;  // locate the last node
5.        Node next = last.next;
 // identify the position to append the new node
6.        if (last == tail) {
7.           if (next == null) {  // no successor
8.              if (CAS(last.next, next, node)) {  // append the new node
9.                 CAS(tail, last, node);   // new node is the tail
10.                return;
11.             }
12.          } else {
 // tail has a successor; another thread is in between 8−9
13.             CAS(tail, last, next);
14.          }
15.       }
16.    }
17. }
```

Lock Free Queue: Enqueue

```
1.  void enq(int value) {
2.      Node node = new Node(value);  //  create a new node
3.      while (true) {
4.          Node last = tail;  //  locate the last node
5.          Node next = last.next;
 //  identify the position to append the new node
6.          if (last == tail) {
7.              if (next == null) {  //  no successor
8.                  if (CAS(last.next, next, node)) {  // append the new node
9.                      CAS(tail, last, node);   // new node is the tail
10.                      return;
11.                 }
12.             } else {
 // tail has a successor; another thread is in between 8−9
13.                 CAS(tail, last, next);   // set tail to correct node
14.             }
15.         }
16.     }
17. }
```

Lock Free Queue: Dequeue

```
int deq() {
  while (true) {
    Node first = head;
    Node last = tail;
    Node next = first.next;
    if (first == head) {
      if (first == last) {
        if (next == null) {
          return EMPTY;
        }
        CAS(tail, last, next);
      } else {
        int value = next.value;
        if (CAS(head, first, next))
          return value;
      }
    }
  }
}
```

Lock Free Queue: Dequeue

```
int deq() {
   while (true) {
      Node first = head;
      Node last = tail;
      Node next = first.next;
      if (first == head) {
         if (first == last) {
            if (next == null) {
               return EMPTY;
            }
            CAS(tail, last, next);   // update the tail
         } else {
            int value = next.value;
            if (CAS(head, first, next))
               return value;
         }
      }
   }
}
```

```
int deq() {
   while (true) {
      Node first = head;
      Node last = tail;
      Node next = first.next;
      if (first == head) {
         if (first == last) {
            if (next == null) {
               return EMPTY;
            }
            CAS(tail, last, next);   // update the tail
         } else {
            int value = next.value;
            if (CAS(head, first, next))   // move head
               return value;
         }
      }
   }
}
```

ABA Problem

Thread T1 reads value A from shared memory,

Thread T1 is preempted, allowing process T2 to run

Thread T2 modifies the shared memory value A to value B to value A

Thread T1 begins starts, observes that the shared memory value is unchanged and continues

Example: ABA Problem

Queue: Head $\rightarrow$ a $\Rightarrow$ b $\Rightarrow$ c $\leftarrow$ tail
Step (1): T1.deq(): first = a, next = b
Step(2): T2 removes a and b from the queue

Queue: Head $\rightarrow$ c $\leftarrow$ tail
T1.deq(): first = a, next = b

Step(3): Node a is enqueued back to the queue

Queue: Head $\rightarrow$ a $\Rightarrow$ c $\leftarrow$ tail
T1.deq(): first = a, next = b

Step(4) T1 perform CAS successfully

Head is now pointing to b which is not in queue

Solution

Tag/mark the memory accesses

AtomicStampedReference in Java

Hardware: Load-Linked/Store-Conditional (LL/SC)

$\mathbb{P}: \mathsf{LL}(x, v); \mathsf{cmp}; \mathsf{ne}\ \mathbb{Q}; \mathsf{SC}(x, nw\_v); \mathsf{teq}\ \mathbb{P}; \mathbb{Q}:$

SC is performed if x is not accessed after LL.

LL/SC instructions are available in various architectures e.g. ARM, Power

CMPXCHG in x86 does not suffice

Safety property:

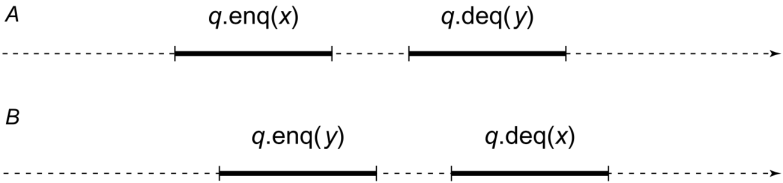
(1) Map the concurrent executions to sequential executions

(2) Reason about these sequential execution executions

Prove that the object satisfies a given sequential specification

- $\{q\}enq(x)\{q.x\}$
- $\{a.q\}deq()\{q\}$
- $\{\epsilon\}deq()\{EMPTY\}$

Sequential consistent Execution:

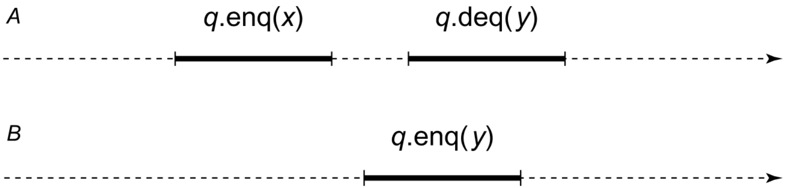


Interleaving executions:

$q.enq(x).q.enq(y).q.deq(x).q.deq(y)$

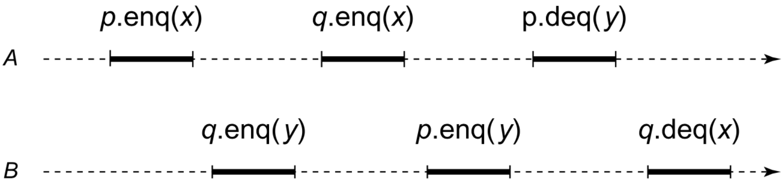
$q.enq(y).q.enq(x).q.deq(y).q.deq(x)$

Sequential consistent execution but does not follow time-order



Execution

Drawback of sequential consistent execution: not compositional



$p.enq(y) \rightarrow p.enq(x)$
$q.enq(x) \rightarrow q.enq(y)$
$p.enq(x) \rightarrow p.enq(x)$
$q.enq(y) \rightarrow p.enq(y)$

We need stronger executions

Linearizability

Method call = (invoke, response)

Linearizability: each method call should take effect instantaneously at some point during (invoke, response)

- linearizable point

Linearizability is compositional

We derive sequential execution and check if it satisfy sequential specification

Progress Conditions

Liveness properties:

Wait-freedom: Every call finishes its execution in a finite number of steps

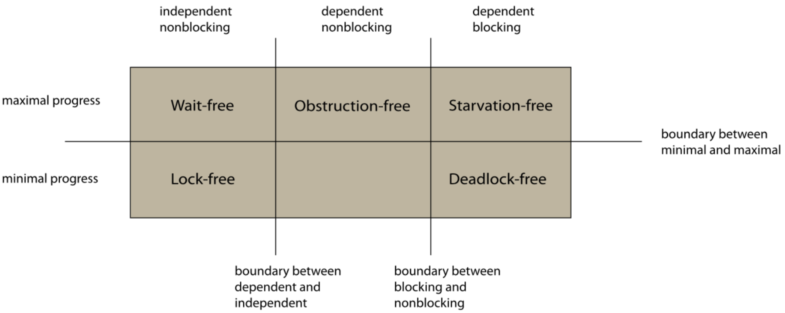
- Non-blocking property
- An arbitrary delay in one thread cannot prevent other threads to make progress

Lock-freedom: Some (rather than all threads) thread makes progress

wait-free $\implies$ lock-free

Obstruction-freedom: a thread makes progress if no other thread interferes

Progress Conditions



Progress properties of the concurrent queue?

Stack

Push():

1. Create a new node
2. Set its next pointer to the current head node.
3. Set the head node to point to it.

$$nw = new\ Node(\ldots);$$
$$nw.next = head;$$
$$head = nw;$$

Pop():

1. Read the current value of head
2. Read *head.next*
3. Set *head* to *head.next*
4. Return the data from the retrieved node
5. Delete the retrieved node

$$top = head;$$
$$n = head.next$$
$$head = n;$$
$$return\ top.data;$$
$$delete(top);$$

Stack: Concurrent push()

1. Create a new node
2. Set its next pointer to the current head node.
3. Set the head node to point to it.

Sequential Version:

$$nw = new\ Node(\ldots);$$
$$nw.next = head;$$
$$head = nw;$$

Concurrent Version:

$$\begin{array}{l|l} nw = new\ Node(\ldots); & nw = new\ Node(\ldots); \\ nw.next = head; & nw.next = head; \\ while(\neg CAS(head, nw.next, nw)) & while(\neg CAS(head, nw.next, nw)) \\ \quad ; & \quad ; \end{array}$$

Stack: Concurrent push()

1. Create a new node
2. Set its next pointer to the current head node.
3. Set the head node to point to it.

Concurrent Version:

$$nw = new\ Node(\ldots);$$
$$nw.next = head;$$
$$while(\neg CAS(head, nw.next, nw))$$
$$;$$

Multiple push() operations:

- Is it possible to insert multiple elements in the same stack slot?
- Any possibility of deadlock?

Stack: Concurrent Pop()

1. Read the current value of head
2. Read *head.next*
3. Set head to *head.next*
4. Return the data from the retrieved node
5. Delete the retrieved node

Concerns for concurrent Pop():

Execution. T1: 1, T2: 1, T1: 2-5, T2: 2-5

Error: T2 accesses dangling pointer

Stack: Concurrent Pop()

1. Read the current value of head
2. Read *head.next*
3. Set head to *head.next*
4. Return the data from the retrieved node
5. Delete the retrieved node

Concerns for concurrent Pop():

Execution. T1: 1, T2: 1, T1: 2-5, T2: 2-5

Error: T2 accesses dangling pointer

Solution (for now): No step 5

Stack: Concurrent Pop()

1. Read the current value of head
2. Read *head.next*
3. Set head to *head.next*
4. Return the data from the retrieved node
5. Delete the retrieved node

Concerns for concurrent Pop():

Execution. T1: 1, T2: 1, T1: 2-5, T2: 2-5

Error: T2 accesses dangling pointer

Solution (for now): No step 5

New problem: T1 & T2 returns same value

Stack: Concurrent Pop()

1. Read the current value of head
2. Read *head.next*
3. Set head to *head.next*
4. Return the data from the retrieved node
5. Delete the retrieved node

Concerns for concurrent Pop():

Execution. T1: 1, T2: 1, T1: 2-5, T2: 2-5

Error: T2 accesses dangling pointer

Solution (for now): No step 5

New problem: T1 & T2 returns same value

Solution: use CAS

Pop():

1. Read the current value of head
2. Read $head.next$
3. Set head to $head.next$
4. Return the data from the retrieved node
5. Delete the retrieved node

$top = head;$
$while(\neg CAS(head, top, top.next))$
$;$
$return\ top.data;$

If CAS succeeds then (4) Return the data from the retrieved node

Failed CAS: some other thread has performed Push()/Pop()

Stack: Concurrent Pop()

Pop():

1. Read the current value of head
2. Read *head.next*
3. Set head to *head.next*
4. Return the data from the retrieved node
5. Delete the retrieved node

$$
\begin{aligned}
&top = head;\\
&while(\neg CAS(head, top, top.next))\\
&\quad ;\\
&return\ top.data;
\end{aligned}
$$

Problems:

1. Does not work for empty queue
2. Step 5 is required to delete the popped node

Solution for the empty queue

$$
\begin{array}{l}
top = head; \\
while(top\ \&\&\ \neg CAS(head, top, top.next)) \\
\quad ; \\
return\ top.data;
\end{array}
$$

References

The Art of Multiprocessor Programming (chapter 3, 10)
2nd Edition
Authors: Maurice Herlihy, Nir Shavit, Victor Luchangco, Michael Spear

C++ Concurrency in Action (Chapter 7)
Practical Multithreading
Anthony Williams