CSC2/458 Parallel and Distributed Systems
Parallel Data Structures - II

Sreepathi Pai

February 27, 2018

URCS

Outline

Parallelizing a Counter

Some Non-Blocking Data Structures

Outline

A Counter

```
class SerialCounter:
    int counter_value

    def add(n):
        counter_value += n

    def get_count():
        return counter_value
```

A Parallel Counter with Locks

```
class ParallelCounterLocks:
    int counter_value
    lock cv

    def add(n):
        cv.lock()
        counter_value += n
        cv.unlock()

    def get_count():
        cv.lock()
        return counter_value
        cv.unlock()
```

A Parallel Counter without Locks

```
class ParallelCounterNoLocks:
    int counter_value

    def add(n):
        atomic_add(&counter_value, n)
        write_to_all_fence()

    def get_count():
        return counter_value
```

A Faster Parallel Counter without Locks

```
class ParallelCounterNoLocks:
    int counter_value

    int thread_local_adds[nthreads] = {0}

    def add(n):
        thread_local_adds[current_thread_id] += n

    def get_count():
        old = atomic_add(&counter_value, thread_local_adds[self])
        v = old + thread_local_adds[self]
        thread_local_adds[self] = 0
        return v
```

The semantics of ParallelCounterNoLocks

- Operations in a thread happen in order for a counter
- Operations from a different thread are not visible to other threads until that thread calls `get_count`

What are the semantics of ParallelCounterNoLocks?

- Is this equivalent to a sequential counter semantics?
- Why is it not sequentially consistent?

Semantics of Concurrent Objects

Ideally,

- Concurrent objects should exhibit "behavioral" equivalence to their sequential counterparts
- It should always be possible to find an ordering of operations on a single concurrent data object
- It should always be possible to compose these individual orders into a total order of operations across all concurrent objects
- These orderings should be "intuitive"

Why should we follow these semantic requirements?

Fulfilling would allow reasoning about the parallel program as if it were a sequential program:

- Construct the total order of operations in the parallel program
- Compare the results of the parallel program with the sequential semantics of the program

Review: Sequential Consistency for Concurrent Objects

- If operations on a concurrent object
  - Appear to happen in some serial, interleaved order across program threads
  - While respecting program order within a thread
- Then that object is sequentially consistent

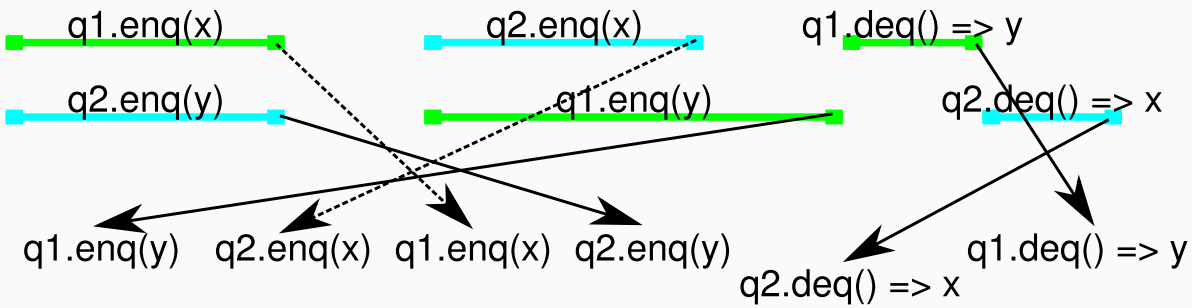
Review: Two sequentially consistent queues

```
T0:                 T1:
a: q1.enq(x)        x: q2.enq(y)
b: q2.enq(x)        y: q1.enq(y)
c: q1.deq() => y    z: q2.deq() => x
```

Ordering rules:

- $y \rightarrow a$ (implied by c)
- $b \rightarrow x$ (implied by z)
- $a \rightarrow b$ (thread order)
- $x \rightarrow y$ (thread order)

Why did this happen?



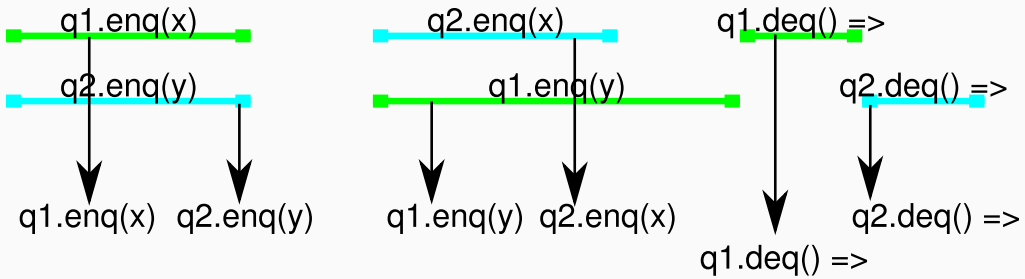
Sequential consistency allows operations on different concurrent objects to execute out-of-order within the same thread. One example is highlighted by dashed lines.

Adding some requirements

The effects of an operation must appear to take place:

- instantaneously at a point ...
- ... between its call and return

Timeline with additional requirements



Note, the requirements now imply the following ordering:

- `q1.enq(x)` happens before `q2.enq(x)`
- `q2.enq(y)` happens before `q1.enq(y)`
- Any interleaving that respects this ordering will not allow `q1.deq() => y` and `q2.deq() => x`
  - But may allow other orders (hence I've left the return values in the figure empty)

Linearizability

- By adding a "real-time" ordering requirement to sequential consistency, we can build "linearizable" objects
- Linearizability is composable
  - See Herlihy and Wing (TOPLAS ACM Trans. Program. Lang. Syst. 12, 3 (July 1990), 463-492) for details
- Key practical questions:
  - How to achieve "instantaneous" execution?
  - How to determine total order for reasoning?

Outline

A Parallel Counter with Locks

```
class ParallelCounterLocks:
    int counter_value
    lock cv

    def add(n):
        cv.lock()
        counter_value += n
        cv.unlock()

    def get_count():
        cv.lock()
        return counter_value
        cv.unlock()
```

- Will a thread ever wait for another thread?
  - Yes. In fact, if a thread that is in the critical section is pre-empted, other threads will wait without progressing.
  - This is a *blocking* counter.

A Parallel Counter with Atomic Add

```
class ParallelCounterNoLocks:
    int counter_value

    def add(n):
        atomic_add(&counter_value, n)
        write_to_all_fence()

    def get_count():
        return counter_value
```

- Will a thread ever wait for another thread?
  - No.
  - This is a *non-blocking*, *wait-free* counter.

A Parallel Counter with CAS

```
class ParallelCounterNoLocks:
    int counter_value

    def add(n):
        do {
           old = counter_value
           new = old + n
        while(atomic_CAS(&counter_value, old, new) != old);
        write_to_all_fence()

    def get_count():
        return counter_value
```

- Will a thread ever wait for another thread?
  - No, but a thread may starve (it may never succeed in the `atomic_CAS`)
  - But some thread will make progress
  - This is a *non-blocking, lock-free* counter.

Progress Guarantees

- Blocking data structures
  - These use locks
- *Progress*: operations complete in a finite number of steps
- Non-blocking data structures
  - Wait-free: The system makes progress
  - Lock-free: Some thread makes progress
  - Obstruction-free: A thread will make progress if not obstructed

A Non-blocking Stack with the ABA problem

```
node* pop(node** top):
    node* old, new
    repeat
       old := *top
       if old = null return null
       new := old->next
    until CAS(top, old, new)
return old
```

The Treiber Stack with Counted Pointers

```
node* stack.pop()
repeat
   <o, c> := top
   if o = null return null
   n := o->next
until CAS(&top, <o, c> , <n, c+1>)
```

The M&S Queue

Handout: Pg 126 of MLS