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

Sreepathi Pai

January 18, 2018

URCS

Outline

Concurrent Objects

Correctness/Safety

Outline

Concurrent Objects

Correctness/Safety

Concurrent Objects/Data Structures

- Like normal objects, except that methods may be called concurrently
  - Implementation is internal
  - Interface (i.e. methods) is public
- Key concerns:
  - Correctness (Safety)
  - Progress (Liveness)
  - Concurrency (Performance)

Per-instance locks

- Each object instance has an associated lock
- Each method acquires this object lock on entry
- Each method releases this object lock on exit
  - Multiple exits must be handled carefully!
- End result: Only one method in the object can be running at a time

Monitors

- Per-method locks are also called "Monitors"
- Ensure that concurrent operations always execute in some serial order
  - The order in which they obtained locks

```
class queue:
     lock qlock

     def enq(item):
         qlock.acquire()
         ...
         qlock.release()

     def deq(item):
         qlock.acquire()
         ...
         qlock.release()
```

Issues with Monitors

- Lack of concurrency
  - Amdahl's law!
- What if no locks were used in implementation?
  - E.g., a lock is a concurrent object which may use just ordinary reads/writes or atomic read-modify-writes

Outline

Concurrent Objects

Correctness/Safety

The tale of the fast concurrent queue

```
T0:                 T1:
   q1.enq(x)           q1.enq(y)
   q1.deq()            q1.deq()
```

- Both `q1.deq()` in both threads returned empty!
  - What kind of implementation allows this?
- What behaviour are we expecting?

A possible implementation

- All `deq` operations are prioritized over `enq` operations
    - Two separate *internal* queues for `deq` and `enq`
    - All commands in `deq` processed before `enq`
- Does this implementation resemble a concurrent data structure you already use?

System Memory as a Concurrent Data Structure

- RAM is a concurrent data structure
  - Supports two methods `read` and `write`
- Many CPUs have separate queues for reads and writes
- Many memory systems may also reorder reads (and/or) writes
- How did we reason about ordering in such systems?

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

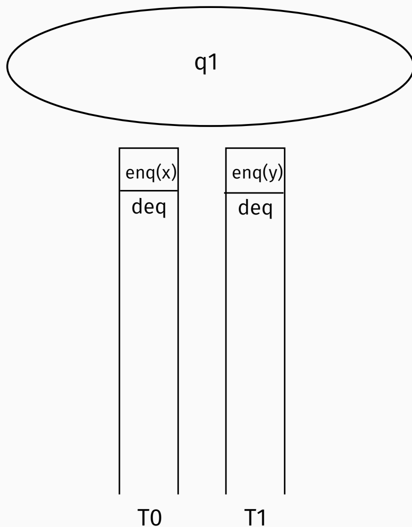
The Sequentially Consistent Queue

Assume q1 is a sequentially consistent queue

```
T0:                  T1:
   q1.enq(x)            q1.enq(y)
   q1.deq()             q1.deq()
```

- What can q1.deq() in T0 return?
- What can q1.deq() in T1 return?
- Can either return empty?

Implementing a Sequentially Consistent Queue

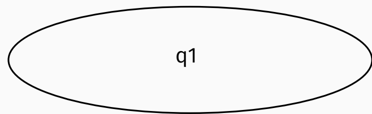


How do two sequentially consistent queues interact?

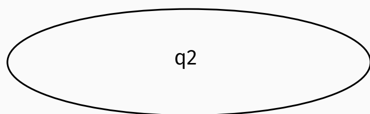
```
T0:                   T1:
   q1.enq(x)             q2.enq(y)
   q2.enq(x)             q1.enq(y)
   q1.deq() => y         q2.deq() => x
```

- Is the value of `q1.deq()` sequentially consistent?
- Is the value of `q2.deq()` sequentially consistent?
- Can you find a single *total* order for operations on q1 and q2 that:
    - follows program order in each thread,
    - in an interleaving of operations across each thread in some serial order?

Composing Two Sequentially Consistent Queues?



enq(x)
deq
enq(y)
T0
T1



enq(x)
enq(y)
deq
T0
T1

Sequential Consistency Does Not Compose

- A system built out of sequentially consistent components may itself not be sequentially consistent!
- Does this matter for RAM?
- How do we build a sequentially consistent system for multiple queues?

Linearizability

- Object has some notion of sequential semantics
  - `enq` places item in queue
  - `deq` removes item from non-empty queue
- For the equivalent concurrent object:
  - Construct a *sequential history* of operations
  - If this history is consistent with sequential semantics, the object is linearizable
- To build linearizable objects
  - Each operation must appear to complete "instantaneously" ...
  - ... and must do so between its call and return
  - (note: calls and returns appear as two separate items in the history)

Translation

- Avoid partial updates
  - Maintain atomicity of all shared data
  - Including multiple variables
- Identify "linearization" points
  - These are points where operation occurs "instantaneously"
  - (where operation's effects are made visible to other threads)
- Order concurrent operations in the history by order of execution of their linearization points

Linearizability is composable

- If operations on two objects are linearizable individually,
- then the operations on the two objects in a program are also linearizable

Linearizability and multiple objects

Assume two bank accounts – A and B – each containing 500 units of money

```
      T0                   T1
A.withdraw(100)     sum = B.balance()
B.deposit(100)      sum += A.balance()
```

- `withdraw`, `deposit` and `balance` are all atomic
- This sequence is linearizable
- Can `sum` ever be 900?

Yes

```
      T0                    T1
A.withdraw(100)
                    sum = B.balance()
                    sum += A.balance()
B.deposit(100)
```

Serializability

- Unit of execution is not operations, but a *transaction*
- If transactions appear to execute one at time in some *total* order, then they *serialize*
- Below, sum will always be 1000, regardless of order of execution of transactions, assuming A and B started with 500

```
      T0                   T1
transaction {
  A.withdraw(100)
  B.deposit(100)
}
                    transaction {
                       sum = B.balance()
                       sum += A.balance()
                    }
```

Implementing Transactions

- Grab locks for A and B at beginning of transaction
- Release locks for A and B at end of transaction
- What is the problem here?

```
      T0                   T1
transaction {
  A.withdraw(100)
  B.deposit(100)
}
                    transaction {
                       sum = B.balance()
                       sum += A.balance()
                    }
```

Two-phase locking and transaction retrying

- Acquiring locks may cause deadlocks
- Two-phase locking
  - Only acquire locks (”expansion”)
  - Only release locks (”contraction”)
- If deadlock detected (e.g. dependence graph between transactions)
  - Release all locks
  - Retry transaction

Transaction Ordering

This paragraph describes, generally, how we post transactions to accounts. Please note that this process may change from time to time, without prior notice to you. Our order of posting depends on a number of factors, including when a transaction occurs, whether it has already been approved by us or has become final, the order in which it presented, the amount, system availability, potential risk of loss to the Bank, and the type of transaction in question, among other variables. Usually, deposits are posted before debits, and checks are posted in order of amount, from low to high. There are several exceptions to this posting order, however. We also generally process previously authorized transactions (e.g., checks cashed at the Bank), wires, transfers, Bank fees and ACH debits before we pay your checks. **We always reserve the right to post transactions that are payable to us first**, and we may post any transaction earlier or later in the process than indicated. As such, if you want to avoid an overdraft or the possibility of a rejected transaction, you should take steps to ensure that your account has sufficient funds to cover each of your transactions and our fees.