Design of Parallel and High Performance Computing
HS 2013
Markus Püschel, Torsten Hoefler
Department of Computer Science
ETH Zurich

Homework 3
Out: 2013-10-17
Revision: 1

Linearizability

Exercise 1

For the following history of a shared register with the operations write(x)/void and read()/x answer the questions below.

B: r.write(1)
A: r.read()
C: r.write(2)
A: r:1
B: r:void
C: r:void
B: r:read()
B: r:1
A: q.write(3)
C: r.read()
A: q:void

- What is $H|B$?
- What is $H|r$?
- Turn H into a complete subhistory H'.
- Is H' sequential?
- Is H' well-formed?
- Is H' linearizable? If yes, prove it!
- If the first two events are swapped, is the resulting history equivalent to H?

Exercise 1 Solution

0.0.1 Thread Subhistory

$H|B$ (H at B) is the thread subhistory of thread B of hitory H. For the given H it is

B: r.write(1)
B: r:void
B: r:read
B: r:1

0.0.2 Object Subhistory

$H|r$ (H at r) is the object subhistory of object r of history H. For the given H it is

B: r.write(1)
A: r.read()
C: r.write(2)

A: r:1
B: r:void
C: r:void
B: r:read()
B: r:1
C: r.read()

0.0.3 Sequential

A history H is sequential if:

1. The first event of H is an invocation.
2. Each invocation, except possibly the last, is immediately followed by a matching response. Each response is immediately followed by an invocation.

Therefore, the given history is not sequential, because the first event is an invocation, but it is followed by another invocation, not by a matching response. A history which is not sequential is concurrent.

0.0.4 Well-formed

A history is well-formed if each thread subhistory $H|P$ of H is sequential. For the given H, the thread subhistories are

A: r.read()
A: r:1
A: q.write(3)
A: q:void

B: r.write(1)
B: r:void
B: r:read()
B: r:1

C: r.write(2)
C: r:void
C: r.read()

All of them are sequential, therefore H is well-formed.

0.0.5 Linearizable

A history is linearizable if it can be reordered into a sequential history, which is correct according to the sequential definition of the object. If a response preceded an invocation in the original history, it must also precede it in the sequential reordering.

Therefore we can reorder the history as shown, which is sequential and observes the semantics of a register (last written value is stored).

C: r.write(2)
C: r:void
B: r.write(1)
B: r:void
A: r.read()
A: r:1
B: r:read()
B: r:1
A: q.write(3)
A: q:void
C: r.read()

0.0.6 Equivalent

Two histories H and H' are eqivalent, if, for every process P $H|P = H'|P$, which is the case if we swap the first two events in the given history.

Exercise 2

In the following history, do the marked method executions overlap? Or does the method invocation denoted by bold events precede the one which is underlined?

A: q.enq(x)
B: q.enq(y)
A: q:void
B: q:void
B: q.deq()
A: q.deq()
B: q:x

0.1 Exercise 2 Solution

The method invocations do not overlap, as the response **A: q:void** happens before the invocation B: q.deq().

Exercise 3

Is the following history of a fifo queue with the operations enq(x)/void deq()/x linearizable? If yes, prove it! Is it sequentially consistent?

A: r.enq(x)
A: r:void
B: r.enq(y)
A: r.deq()
B: r:void
A: r:y

Exercise 3 Solution

This history is not linearizable because it violates the semantics of the fifo queue. The enqueueing of x precedes the enqueueing of y, therefore x should also be dequeued first. However, the first dequeue operation retrieves y.

It is sequentially consistent however, as sequential consistency does not require the original event precedence to be preserved.

Exercise 4

Is the following history of a fifo queue with the operations enq(x)/void deq()/x linearizable? If yes, prove it!

A: q.enq(x)
B: q.enq(y)
A: q:void
B: q:void
A: q.deq()
C: q.deq()
A: q:y
C: q:y

Exercise 4 Solution

The given history is not linearizable, because the element y is enqueued only once, but dequeued twice. That violates the semantics of a fifo queue.

Parallel FIFO Queue Implementation

Implement a queue with fifo semantics in C or C++, using either OpenMP or POSIX threads. Use locks to implement the queue. Benchmark your implementation on your own machine. Make a diagram out of your benchmark results.