CSE 511: Operating Systems Design

Lectures 21

Linearizability, Sequential Consistency
Progress Conditions

The slides are partially based on Maurice Herlihy's slides from "The Art of Multiprocessing"

Sequential Histories

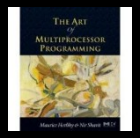
```
A q.enq(3)
A q:void
B p.enq(4)
B p:void
B q.deq()
B q:3
A q:enq(5)
```

match

match

Final pending invocation OK

Method calls of different threads do not interleave



Well-Formed Histories

Per-thread projections sequential

H=

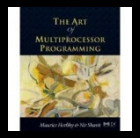
```
A q.enq(3)
B p.enq(4)
B p:void
B q.deq()
A q:void
B q:3
```

H|B=

```
B p.enq(4)
B p:void
B q.deq()
B q:3
```

H|A=

```
A q.enq(3)
A q:void
```



Equivalent Histories

Threads see the same thing in both

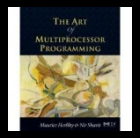
$$\begin{cases} H|A = G|A \\ H|B = G|B \end{cases}$$

H=

```
A q.enq(3)
B p.enq(4)
B p:void
B q.deq()
A q:void
B q:3
```

G=

```
A q.enq(3)
A q:void
B p.enq(4)
B p:void
B q.deq()
B q:3
```



Sequential Specifications

A sequential specification is some way of telling whether

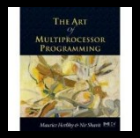
A single-thread, single-object history

Is legal

For example:

Pre and post-conditions

But any style will do …

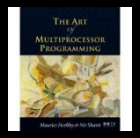


Legal Histories

A sequential (multi-object) history H is *legal* if

For every object *x*

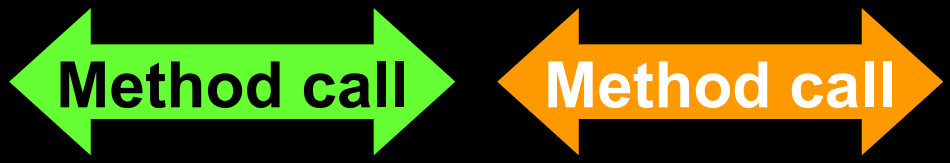
H|x is in the sequential spec for *x*

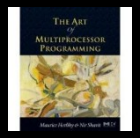


Precedence

```
A q.enq(3)
B p.enq(4)
B p.void
A q:void
B q.deq()
B q:3
```

A method call *precedes* another if response event precedes invocation event

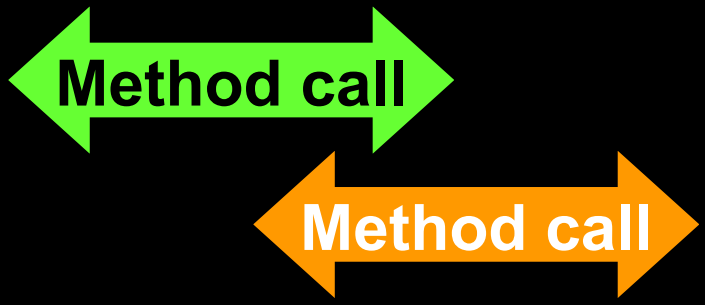


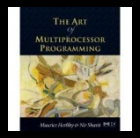


Concurrency

```
A q.enq(3)
B p.enq(4)
B p.void
B q.deq()
A q:void
B q:3
```

Some method calls overlap one another





Linearizability

History H is *linearizable* if it can be extended to G by

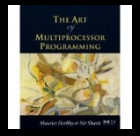
Appending zero or more responses to pending invocations

Discarding other pending invocations

So that G is equivalent to

Legal sequential history S

where S respects “real-time order” of G

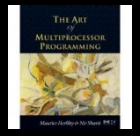


Remarks

Some pending invocations

Took effect, so keep them

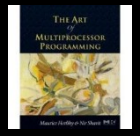
Discard the rest



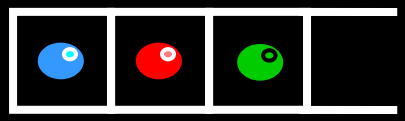
Linearizability: Summary

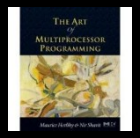
Standard notion of correctness

Captures the notion of objects being “atomic”

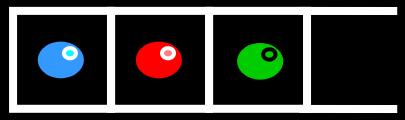


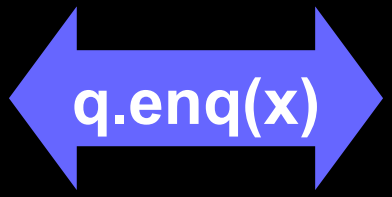
Sequential Consistency: Example

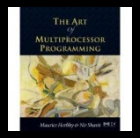




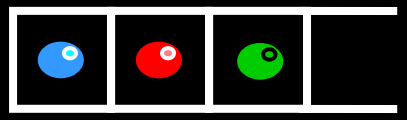
Example







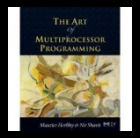
Example



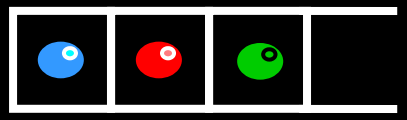
q.enq(x)

q.deq(y)

time



Example

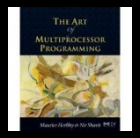


q.enq(x)

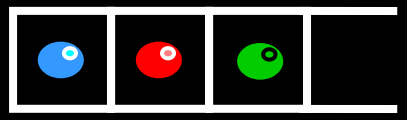
q.deq(y)

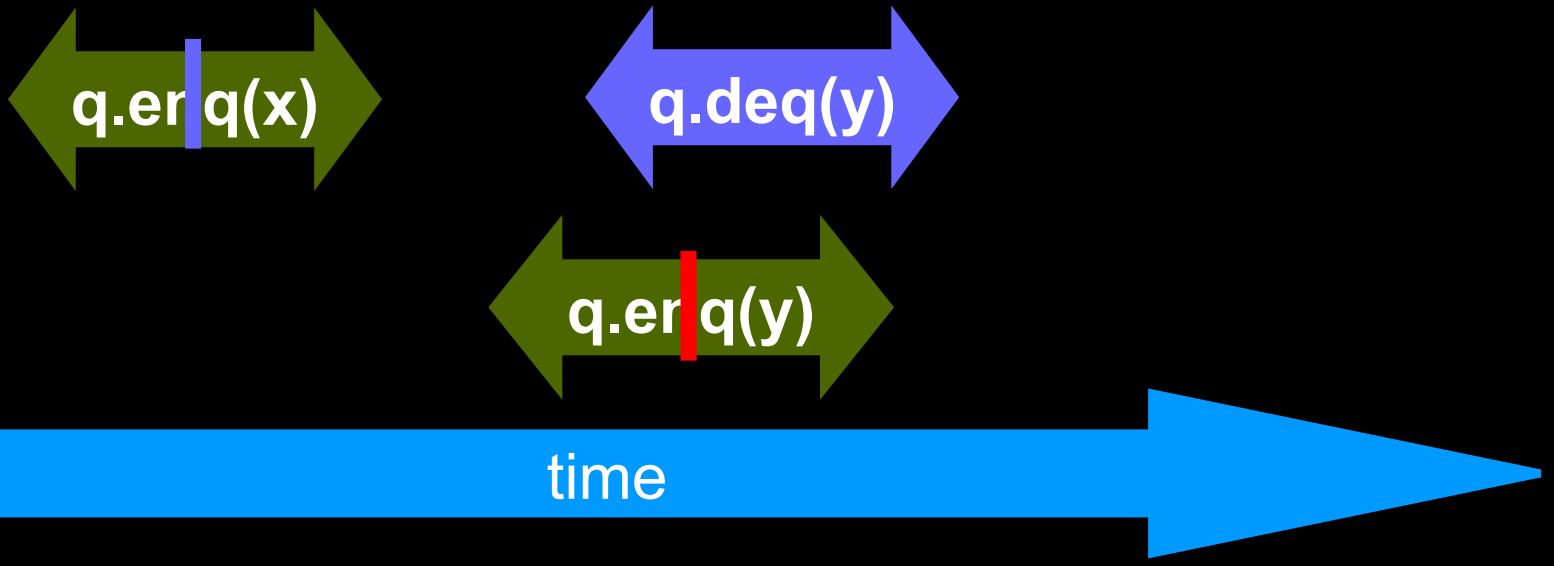
q.enq(y)

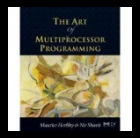
time



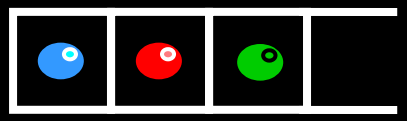
Example







Example



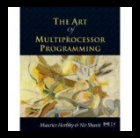
q.enq(x)

q.deq(y)

q.enq(y)

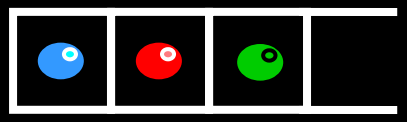
not linearizable

time



Exa

Yet Sequentially Consistent

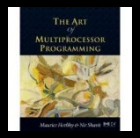


q.enq(x)

q.deq(y)

q.enq(y)

time



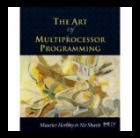
Sequential consistency

Most hardware does not even support that, architects think that sequential consistency is *too strong*

Too expensive for modern hardware

OK if violated by *default*

Honored by *explicit request*



Hardware Consistancy

Initially, a = b = 0.

Core 0

```
mov 1, a     ;Store
mov b, %ebx  ;Load
```

Core 1

```
mov 1, b     ;Store
mov a, %eax  ;Load
```

What are the possible values of %eax and %ebx registers after both processors have executed?

Sequential consistency implies that no execution ends with %eax = %ebx = 0

Hardware Consistency

No modern-day processor implements sequential consistency

Hardware actively reorders instructions.

Compilers too

Because performance is dominated by single-thread unsynchronized execution

Instruction Reordering

```
mov 1, a      ;write
mov b, %ebx   ;read
```

```
mov b, %ebx   ;read
mov 1, a      ;write
```

Program Order

Execution Order

Q. Why might the hardware or compiler decide to reorder these instructions?

A. Higher performance by covering load latency — *instruction-level parallelism*.

Slide used with permission of
Charles E. Leiserson

Instruction Reordering

```
mov 1, a      ;write
mov b, %ebx   ;read
```

```
mov b, %ebx   ;read
mov 1, a      ;write
```

Program Order

Execution Order

Q. When is it safe for the hardware or compiler to perform this reordering?

A. When a ≠ b.

A′. And there's no concurrency.

Slide used with permission of
Charles E. Leiserson

X86: Memory Consistency – TSO (Total Store Ordering)

Thread Code

- Loads are *not* reordered with loads.
- Stores are *not* reordered with stores.
- Stores are *not* reordered with prior loads.
- A load *may* be reordered with a prior store to a different location but *not* with a prior store to the same location.
- Stores to the same location respect a global total order.

Art of Multiprocessor Programming

Memory Barriers (Fences)

A *memory barrier* (or *fence*) is a hardware operation that enforces ordering between the instructions before and after the fence

A memory barrier can be an explicit instruction (x86: mfence)

The typical cost of a memory fence is comparable to that of an L2-cache access

X86: Memory Consistency

Thread's Code

```
Store1
Store2
Load1
Load2
Store3
Store4
Barrier
Load3
Load4
Load5
```

Loads are *not* reordered with loads.

Sto

Sto

loa

A lo

store to

with a prior store to the same location.

Stores to the same location respect a global total order.

Total Store Ordering + properly placed memory barriers = sequential consistency

Memory Barriers

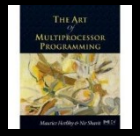
Memory barrier instructipn will

Flush write buffer

Bring caches up to date

Compilers often do this for you

Entering and leaving critical sections

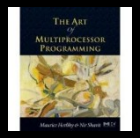


Summary: Real-World

Hardware *weaker* than sequential consistency

Can get sequential consistency at a price

Linearizability better fit for high-level software

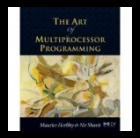


Linearizability

Operation takes effect instantaneously between invocation and response

Based on sequential spec

"gold standard" for concurrent data structures



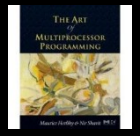
Progress Conditions

Deadlock-free: some thread trying to acquire the lock eventually succeeds

Starvation-free: every thread trying to acquire the lock eventually succeeds.

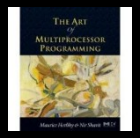
Lock-free: some thread calling a method eventually returns.

Wait-free: every thread calling a method eventually returns.



Progress Conditions

	Non-Blocking	Blocking
Everyone makes progress	Wait-free	Starvation-free
Someone makes progress	Lock-free	Deadlock-free



Kinds of Architectures

SISD (Uniprocessor)

Single instruction stream

Single data stream

SIMD (GPU)

Single instruction

Multiple data

MIMD (Multiprocessors)

Multiple instruction

Multiple data

Kinds of Architectures

SISD (Uniprocessor)

Single instruction stream

Single data stream

SIMD (GPU)

Single instruction

Multiple data

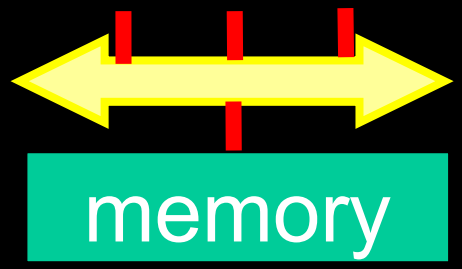
Our space

MIMD (Multiprocessors)

Multiple instruction

Multiple data

MIMD Architectures



Shared Bus **Distributed**

Memory Contention

Communication Contention

Communication Latency