Lecture 13

- Administration

Linearizability - formally

- History H is linearizable if it can be extended to history G so that G is equivalent to legal sequential history S where $\rightarrow_G \subset \rightarrow_S$.

- G is the same as H but without pending invocations:
  - append responses to pending invocations.
  - discard pending invocations.

Linearizability - formally

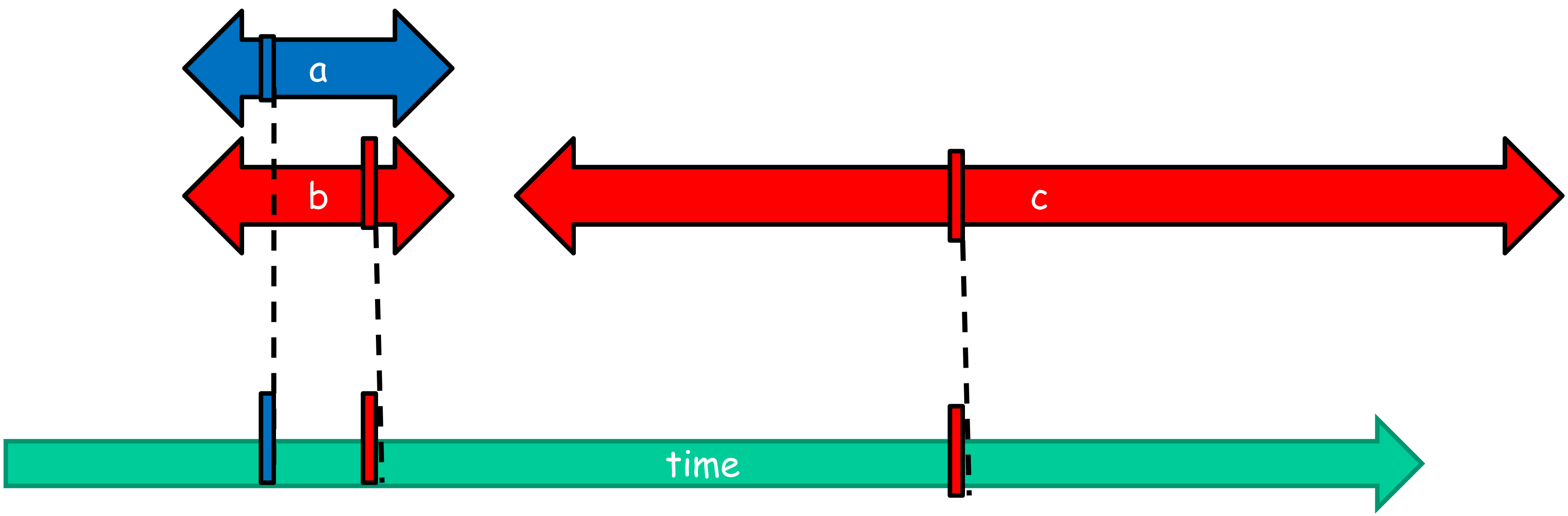
Let's explain what is $\rightarrow_G \subset \rightarrow_S$.

Example:

$\rightarrow_G = \{a \rightarrow c, b \rightarrow c\}$

$\rightarrow_S = \{a \rightarrow b, a \rightarrow c, b \rightarrow c\}$

closure



Example:

Add response to this pending invocation:

Discard this pending invocation:

H =

A q.enq(3)

B q.enq(4)

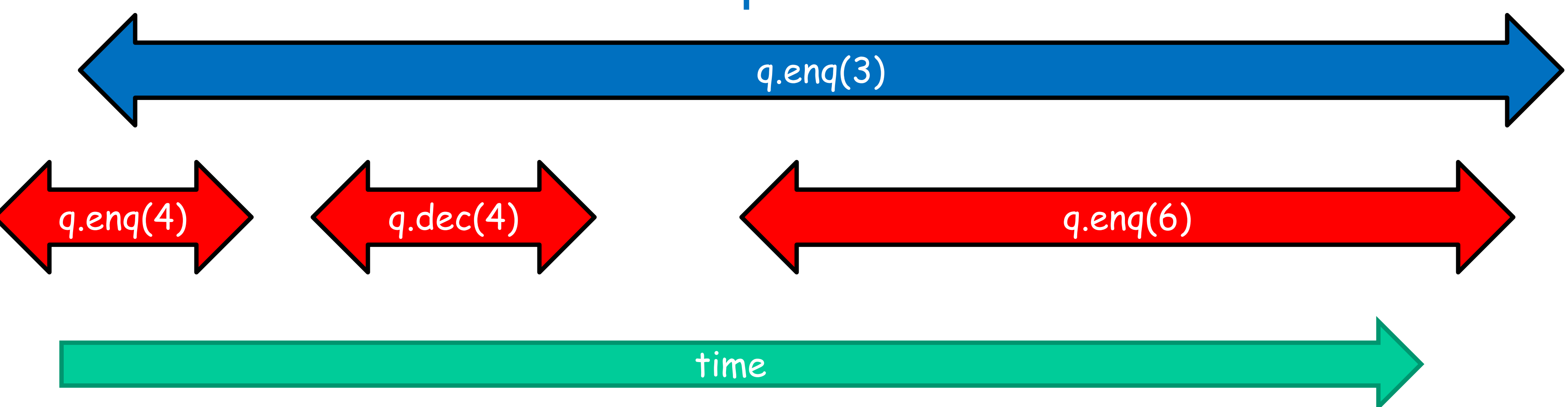
B q:void

B q.deq()

B q:4

B q.enq(6)

A q:void

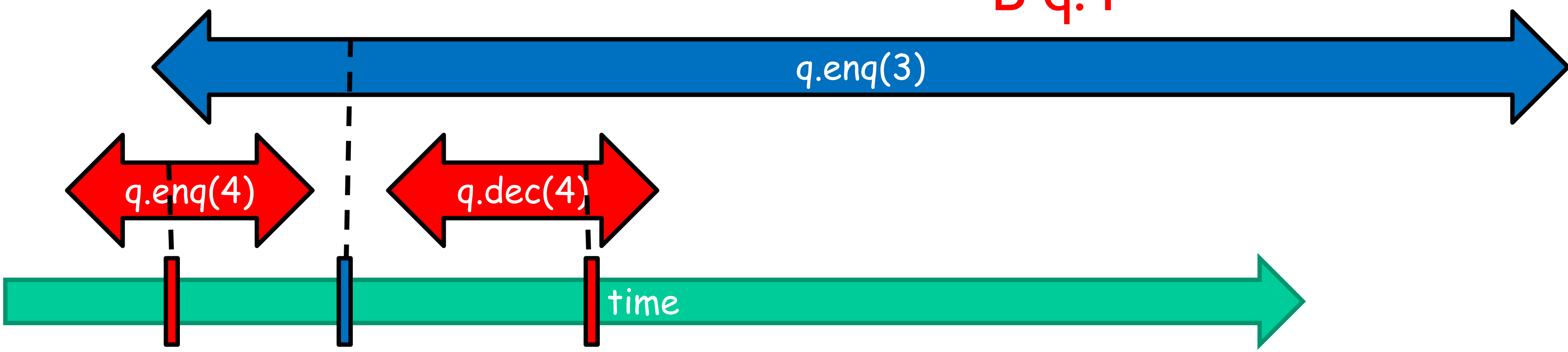


Example (cont'):

G = A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4
A q:void

The equivalent sequential history:

S = B q.enq(4)
B q:void
A q.enq(3)
A q:void
B q.deq()
B q:4



Composability

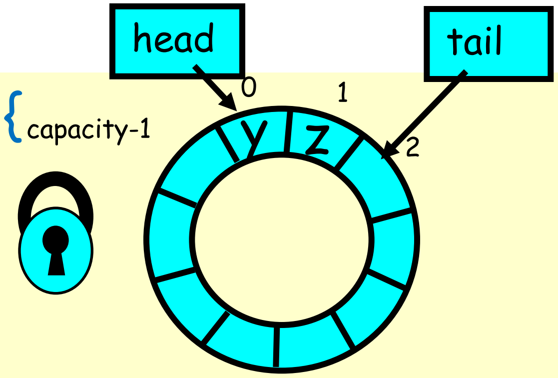
- Linearizability also gives us composability:
  - If we want to construct a new object from linearizable objects, we can be sure that our new object is linearizable too.
- why is it good?
  - It gives us modularity. We can prove linearizability independently for each object.

Linearizability: Summary

- Powerful specification tool for shared objects
- Allows us to capture the notion of objects being “atomic”
- Don’t leave home without it

Reasoning About Linearizability: Locking

```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
      throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```

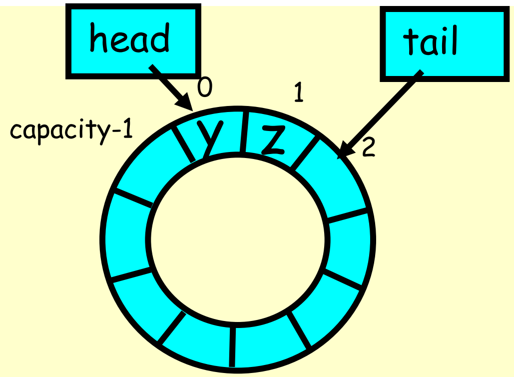


Reasoning About Linearizability: Locking

```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
       throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```

Linearization points are when locks are released

More Reasoning: Lock-free




```
public class LockFreeQueue {

  int head = 0, tail = 0;
  items = (T[]) new Object[capacity];

  public void enq(Item x) {
    while (tail-head == capacity); // busy-wait
    items[tail % capacity] = x; tail++;
  }
  public Item deq() {
     while (tail == head);     // busy-wait
     Item item = items[head % capacity]; head++;
     return item;
}}
```

More Reasoning

```
public cl...           ...Queue

  in...            = 0;
              ...ew Object[c...

          ...oid enq(Item x) {
    ...ile (tail-head == capacity); // busy-wait
    items[tail % capacity] = x; tail++;
  }
  public Item deq() {
    while (tail == head);     // busy-wait
    Item item = items[head % capacity]; head++;
    return item;
}}
```

Linearization order is order head and tail fields modified

Remember that there is only one enqueuer and only one dequeuer

Alternative: Sequential Consistency

- History H is *Sequentially Consistent* 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 a
  - Legal sequential history S

Differs from linearizability

~~Where $\rightarrow_G \subset \rightarrow_S$~~

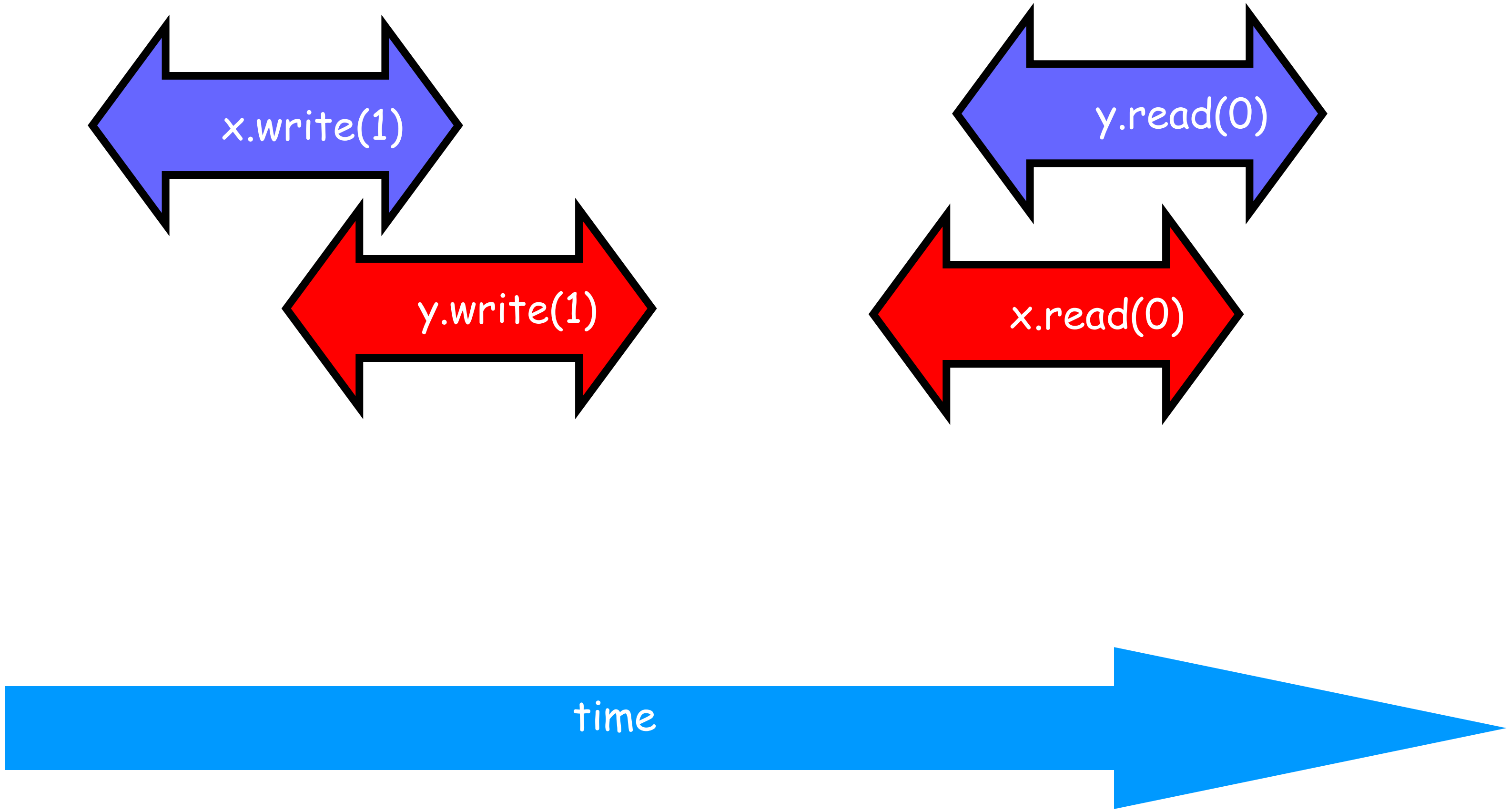
Alternative: Sequential Consistency

- No need to preserve real-time order
  - Cannot re-order operations done by the same thread
  - Can re-order non-overlapping operations done by different threads
- Often used to describe multiprocessor memory architectures

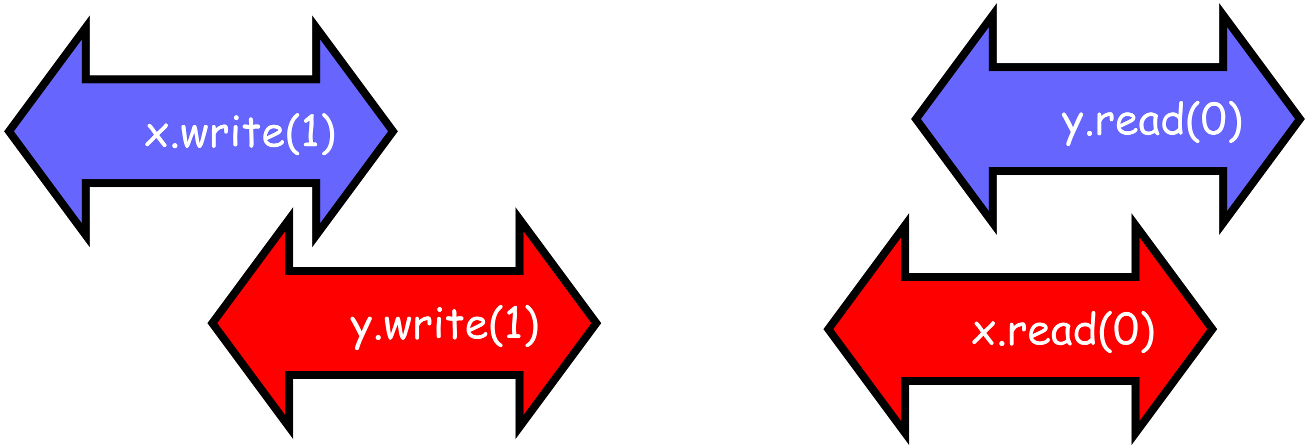
Fact

- Most hardware architectures don't support sequential consistency
- Because they think it's too strong
- Here's another story …

The Flag Example

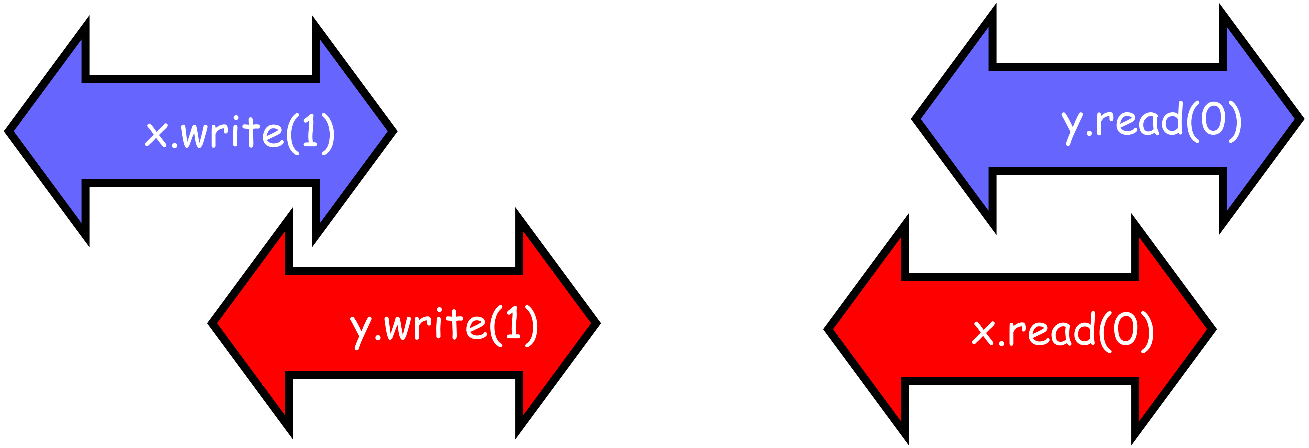


The Flag Example



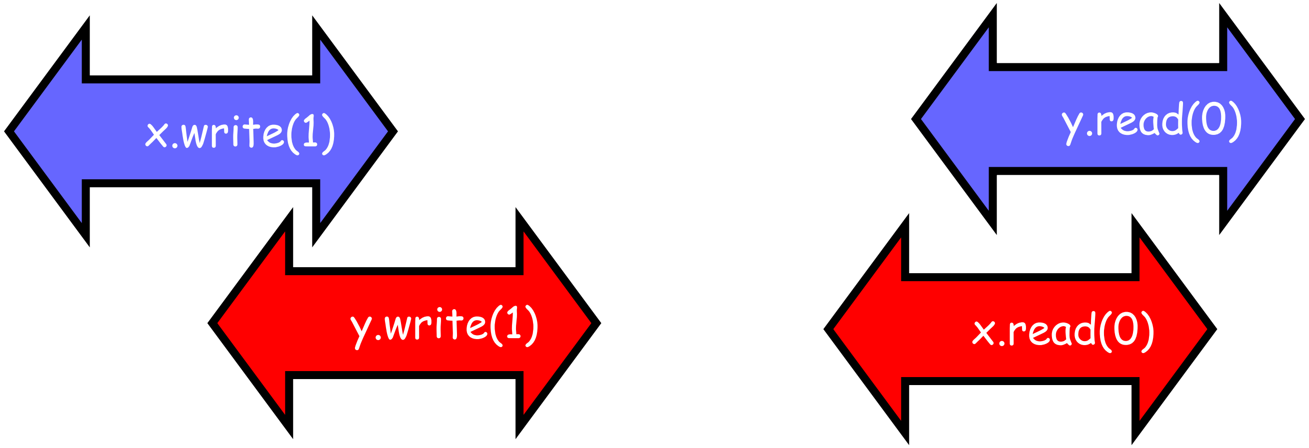
- Each thread's view is sequentially consistent
  - It went first

The Flag Example



- Entire history isn't sequentially consistent
  - Can't both go first

The Flag Example



- Is this behavior really so wrong?
  - We can argue either way …

Opinion1: It's Wrong

- This pattern
  - Write mine, read yours
- Is exactly the flag principle
  - Beloved of Alice and Bob
  - Heart of mutual exclusion
    - Peterson
    - Bakery, etc.
- It's non-negotiable!

Opinion2: But It Feels So Right …

- Many hardware architects think that sequential consistency (with respect to each data memory location) is too strong
- Too expensive to implement in modern hardware
- OK if flag principle
  - violated by default
  - Honored by explicit request

Who knew you wanted to synchronize?

- Writing to memory = mailing a letter
- Vast majority of reads & writes
  - Not for synchronization
  - No need to idle waiting for post office
- If you want to synchronize
  - Announce it explicitly
  - Pay for it only when you need it

Explicit Synchronization

- Memory barrier instruction
  - Flush unwritten caches
  - Bring caches up to date
- Compilers often do this for you
  - Entering and leaving critical sections
- Expensive

Volatile

- In Java, can ask compiler to keep a variable up-to-date with volatile keyword
- Also inhibits reordering, removing from loops, & other “optimizations”

Real-World Hardware Memory

- Weaker than sequential consistency
- But you can get sequential consistency at a price
- OK for expert, tricky stuff
  - assembly language, device drivers, etc.
- Linearizability more appropriate for high-level software

Critical Sections

- Easy way to implement linearizability
  - Take sequential object
  - Make each method a critical section
- Problems
  - Blocking
  - No concurrency

Progress

- We saw an implementation whose methods were lock-based (deadlock-free)
- We saw an implementation whose methods did not use locks (lock-free)
- How do they relate?

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

Transactional Memory

- Software Transactional Memory

- Some Intel processors now supporting it.

The Road Ahead

- Concurrent algorithms pose a great challenge
- It is "easy" to write
  - Correct algorithms and let efficiency take care of itself
  - Efficient algorithms and let correctness take care of itself
- But very hard to write correct & efficient algorithms

Systematically…