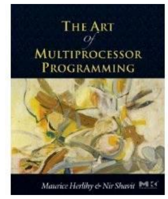
Department of Computer Science

COS 226

Chapter 3
Concurrent objects

Acknowledgement



- Some of the slides are taken from the companion slides for “The Art of Multiprocessor Programming” by Maurice Herlihy & Nir Shavit

Sequential Objects

- In OO programming, an object is a container for data
- Each object has a ***state***
  - Usually given by a set of ***fields***
  - Queue example: sequence of items
- Each object has a set of ***methods***
  - Only way to manipulate state
  - Queue example: **enq** and **deq** methods

Objectivism

- What is a concurrent object?
  - How do we **describe** one?
  - How do we **implement** one?
  - How do we **tell if we’re right**?

Correctness and Progress

- In a concurrent setting, we need to specify both the safety and the liveness properties of an object
  - Safety – nothing bad happens (also known as correctness)
  - Liveness – something good eventually happens (also known as progress)

Sequential objects

- With sequential objects, one way to determine if an object’s methods are behaving correctly is through pre and postconditions.

Sequential Specifications

- If (precondition)
  - the object is in such-and-such a state
  - before you call the method,
- Then (postcondition)
  - the method will return a particular value
  - or throw a particular exception.
- and (postcondition, con't)
  - the object will be in some other state
  - when the method returns,

Pre and PostConditions for Dequeue

- Precondition:
  - Queue is non-empty
- Postcondition:
  - Returns first item in queue
- Postcondition:
  - Removes first item in queue

Pre and PostConditions for Dequeue

- Precondition:
  - Queue is empty
- Postcondition:
  - Throws Empty exception
- Postcondition:
  - Queue state unchanged

Sequential Specifications

- Interactions among methods captured by resulting object state
  - State meaningful between method calls
- Documentation size linear in number of methods
  - Each method described in isolation
- Can add new methods
  - Without changing descriptions of old methods

What About Concurrent Specifications ?

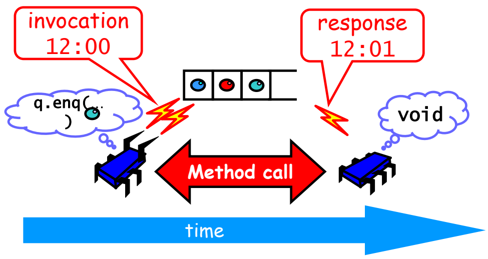
- Methods?
- Documentation?
- Adding new methods?

Correctness and Progress

- Need a way to define
  - when an implementation is correct
  - the conditions under which it guarantees progress

Lets begin with correctness

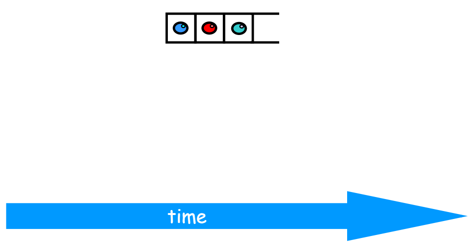
Methods Take Time



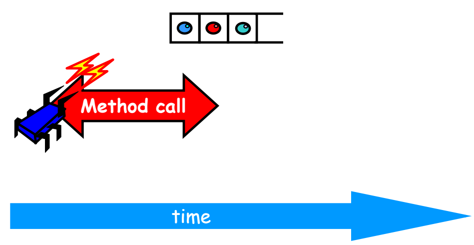
Sequential vs Concurrent

- Sequential
  - Method calls take time? Who knew?
- Concurrent
  - Method call is not an event
  - Method call is an interval.
    - Starts with invocation event
    - Ends with response event
    - Method is pending if invocation has occurred but not yet response

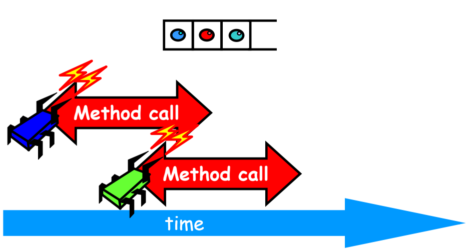
Concurrent Methods Take Overlapping Time



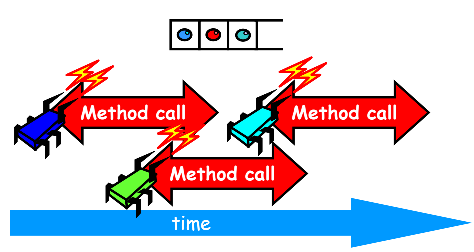
Concurrent Methods Take Overlapping Time



Concurrent Methods Take Overlapping Time



Concurrent Methods Take Overlapping Time



Sequential vs Concurrent

- Sequential:
  - Object needs meaningful state only ***between*** method calls
- Concurrent
  - Because method calls overlap, object might ***never*** be between method calls

Sequential vs Concurrent

- Sequential:
  - Each method described in isolation
- Concurrent
  - Must characterize ***all*** possible interactions with concurrent calls
    - What if two enqs overlap?
    - Two deqs? enq and deq? …

Sequential vs Concurrent

- Sequential:
  - Can add new methods without affecting older methods
- Concurrent:
  - Everything can potentially interact with everything else

Panic!

The Big Question

- What does it mean for a *concurrent* object to be correct?

A Lock-Based Queue

```
class LockBasedQueue<T> {
  int head, tail;
  T[] items;
  Lock lock;
  public LockBasedQueue(int capacity) {
    head = 0; tail = 0;
    lock = new ReentrantLock();
    items = (T[]) new Object[capacity];
}
```

A Lock-Based Queue

```
class LockBasedQueue<T> {
  int head, tail;
  T[] items;
  Lock lock;
  public LockBasedQueue(int capacity) {
    head = 0; tail = 0;
    lock = new ReentrantLock();
    items = (T[]) new Object[capacity];
}
```

Queue fields protected by single shared lock

A Lock-Based Queue

```
class LockBasedQueue<T> {
  int head, tail;
  T[] items;
  Lock lock;
  public LockBasedQueue(int capacity) {
    head = 0; tail = 0;
    lock = new ReentrantLock();
    items = (T[]) new Object[capacity];
}
```

Initially head = tail

A Lock-Based Queue

```
public void enq(T x) throws
  FullException {
   if (tail - head == items.length)
     throw new FullException();
  items[tail] = x;
  tail++;
}
```

Mutual Exclusion?

A Lock-Based Queue

```
public void enq(T x) throws FullException {
  lock.lock();
  try {
     if (tail - head == items.length)
          throw new FullException();
     items[tail] = x;
     tail++;
  } finally {
     lock.unlock();
  }
}
```

Method calls mutually exclusive

Implementation: Deq

```
public T deq() throws EmptyException {
  if (tail == head)
    throw new EmptyException();
  T x = items[head];
  head++;
  return x;
}
```

Implementation: Deq

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

Method calls mutually exclusive

Now consider the following implementation

- The same thing without mutual exclusion
- For simplicity, only two threads
  - One thread enq only
  - The other deq only

Wait-free 2-Thread Queue

```
public class WaitFreeQueue {

  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);
     Item item = items[                    ++;
     return item;
}}
```

How do we define "correct" when modifications are not mutually exclusive?

Read-write example

- Two threads concurrently write -3 and 7 to a register
  - Register – object version of memory location
- Later when another thread accesses the register it returns -7
- Clearly this is wrong – we expect either -3 or 7, but not a mixture

Principle 3.3.1

- Method calls should appear to happen in a one-at-a-time sequential order
  - By itself this principle is too weak to be useful
  - Has to combine it with a stronger condition…

Quiescence

- A object is **quiescent** if it has no pending method calls
  - Can think of it as object is *inactive*

Principle 3.3.2

- **Method calls separated by a period of quiescence should appear to take effect in real-time order**
  - In other words, method calls who are separated by a period of inactivity should appear in the order of their execution
  - Suppose A and B concurrently enqueue x and y, C then enqueues z. We may not be able to predict the order of x and y, but we know they are ahead of z

Quiescent consistency

- Together principle 3.3.1 and 3.3.2 form a correctness property:
  - Quiescent consistency

Quiescent consistency

- An object is quiescent consistent if:
  - Its method calls appear to be in a sequential order
  - Its method calls take place in a real-time order if separated by a period of inactivity

Quiescent consistency

- A shared counter is thus quiescently consistent if:
  - When two concurrent threads write -3 and 7 to a register a later thread will read either -3 or 7 but not a mixture of the two

Quiescent consistency

- Quiescent consistency is **compositional**
  - If each object in the system is quiescent consistent, the whole system will be quiescent consistent.

Another read-write example

- A *single* thread writes 7 and then -3 to a shared register
- Later it reads the register and returns 7
- This is also not acceptable since the value it read is not the last value it wrote

Principle 3.4.1

- Method calls should take effect in program order
  - Program order – The order in which a single thread issues method calls
  - Method calls by different threads are unrelated by program order

Sequential consistency

- Together principles 3.3.1 and 3.4.1 form a second correctness property:
  - Sequential consistency

Sequential consistency

- An object is sequential consistent if:
  - Its method calls are in a sequential order
  - Its method calls are in program order

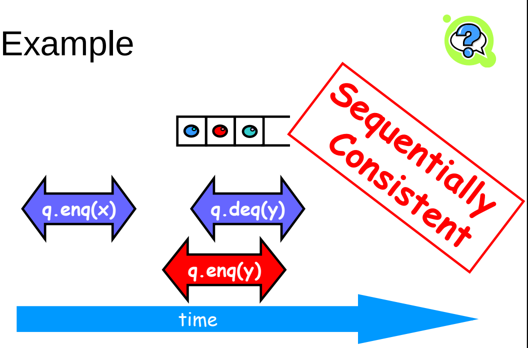
Sequential consistency

- In any concurrent execution, there is a way to order the method calls sequentially so that
  - They are consistent with program order
  - They meet the object's sequential specifications
- There may be more than one order that satisfies these conditions

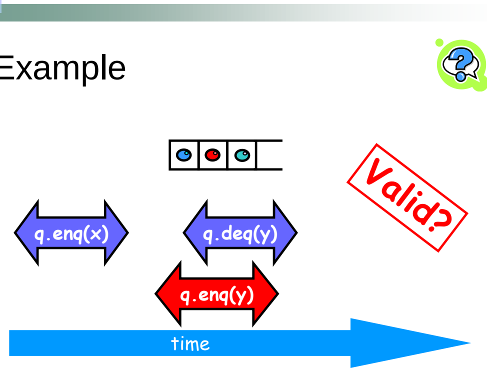
Sequential consistency

- A.enq(x) concurrent with B.enq(y), then A.deq(y) concurrent with B.deq(x)
- Two possible sequential orders:
  - A.enq(x)→B.enq(y)→B.deq(x)→A.deq(y)
  - B.enq(y)→A.enq(x)→A.deq(y)→B.deq(x)
- Both are in program order

Example



Example



Consistency

- Quiescent and sequential consistency are incomparable:
  - The one does not necessarily exist when the other exists
- Quiescent consistency does not necessarily preserve program order
- Sequential consistency is unaffected by quiescent periods

Sequential consistency

- Sequential consistency is not compositional

Principle 3.5.1

- Each method call should appear to take effect instantaneously at some moment between its invocation and response

Linearizability

- Principle 3.5.1 defines a third correctness property:
  - Linearizability
- Each linearizable execution is sequentially consistent, but not vice versa

Linearizability

- Each method should
  - “take effect”
  - Instantaneously
  - Between invocation and response events
- Object is correct if this “sequential” behavior is correct
- Any such concurrent object is
  - **Linearizable™**

Linearizability

- To show that a concurrent object is linearizable one should identify for each method a linearization point where the method takes effect

Linearization points

- For lock-based implementations:
  - Critical section
- For other methods:
  - The single step where the effects of the method call become visible to other methods

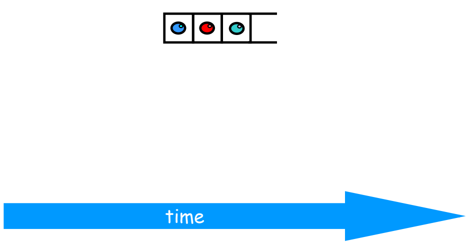
Linearizability

- Sequential consistency is good way to describe standalone systems
- Linearizability is good way to describe components of large system

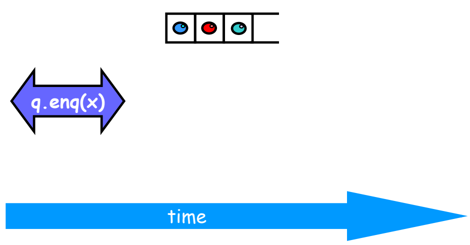
Single-enqueuer/single-dequeuer

- No critical section
- Linearization points depend on execution
- If deq() returns a value:
  - Linearization point = head field is updated
- If list is empty:
  - Linearization point = deq() throws an exception

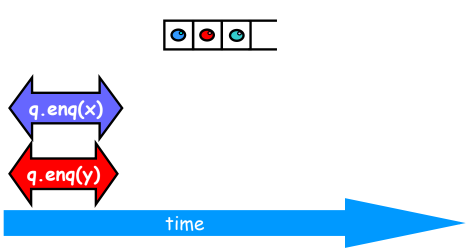
Example



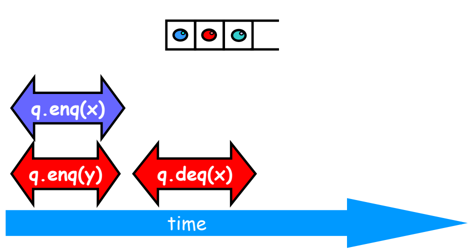
Example



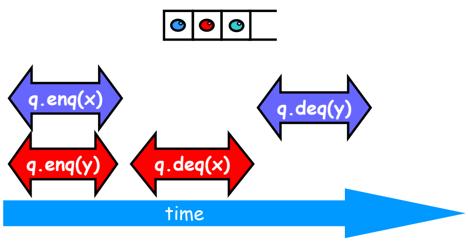
Example



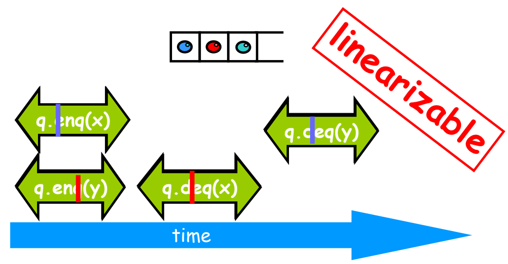
Example



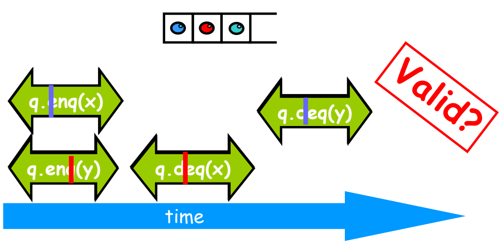
Example



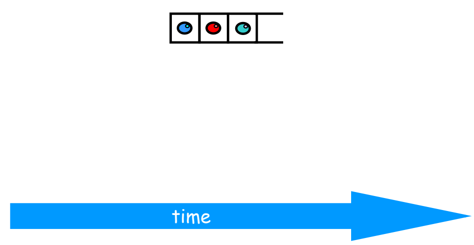
Example



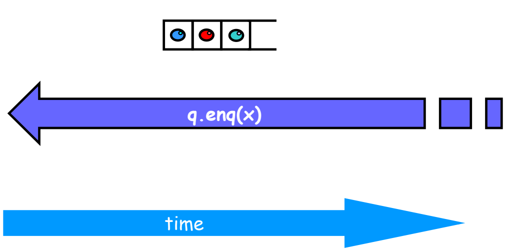
Example



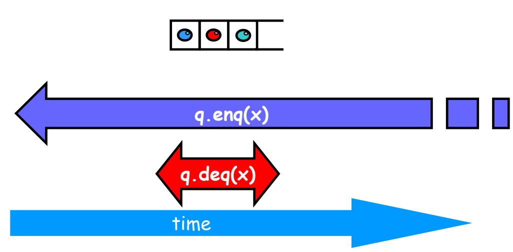
Example



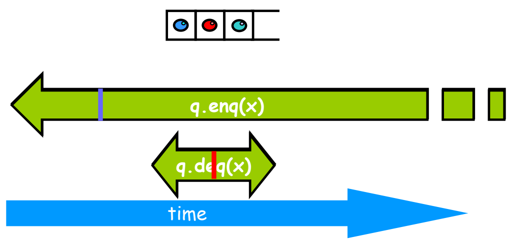
Example



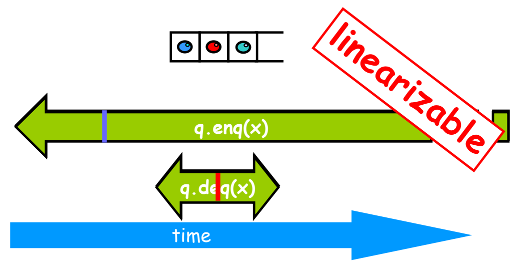
Example



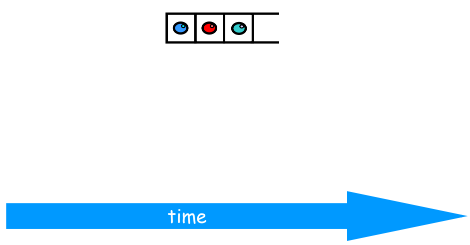
Example



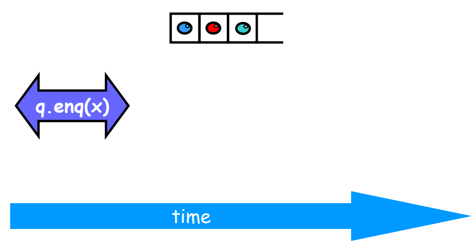
Example



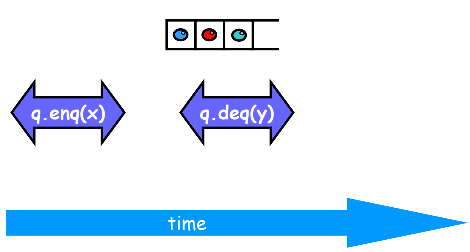
Example



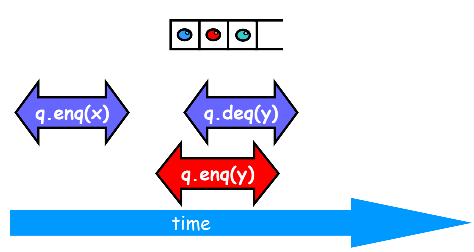
Example



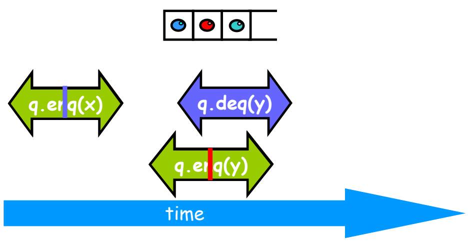
Example



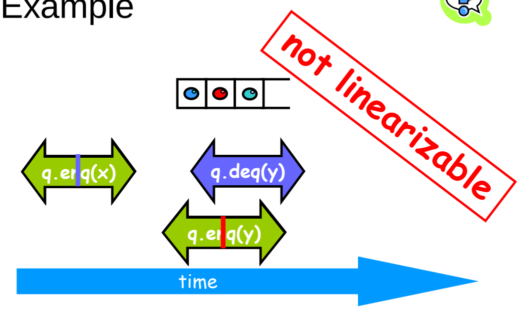
Example



Example



Example



Correctness

- Three correctness conditions:
  - Quiescent consistency
    - Applications that require high performance with weak constraints on object behaviour
  - Sequential consistency
    - Describe low-level systems such as hardware memory interfaces
  - Linearizability
    - Describe higher-level systems composed of linearizable components

Correctness

- Safety property
- Deals with correctness of concurrent execution
  - In correct order?
  - No collisions?

Quiescent consistency

- Checks that method calls appear to be made in sequential order
  - If write 7 and then -3 a read should not be -7
- AND
- Checks that method calls are in real-time order
  - We do not care about the order of concurrent method calls, but when separated by a period of inactivity, method calls should take place in the correct order

Sequential consistency

- Checks that method calls appear to be made in sequential order
  - If write 7 and then -3 a read should not be -7
- AND
- Checks that method calls are made in program order
  - If write 7 and then -3 a read should not be 7

Linearizability

- Checks that method calls appear to take place instantaneously
- Linearization points
  - If one method's linearization point is in the correct program order than a overlapping method, those methods are linearizable

Progress

- Liveness property
- Deals with if different threads have to wait
  - For how long?
  - Will they ever reach the critical section?

Progress

- Progress guarantees can be either:
  - Blocking
    - Delay of any one thread can delay others
  - Non-blocking
    - Delay of one thread cannot delay the others

Lock-free

- A method is lock-free if **some** method calls finishes in a finite number of steps

Wait-free

- A method is wait-free if it **guarantees** that **every call** finishes its execution in a finite number of steps
- It is *bounded* wait-free is there is a limit on the number of steps a method call can take

Lock-free vs. wait-free

- Any wait-free implementation is lock-free, but not vice versa

Lock-free vs. wait-free

- A non-blocking algorithm is:
  - Lock-free if there is guaranteed system-wide progress
  - Wait-free if there is also per-thread progress

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 (succeeds)
- *Wait-free:* every thread calling a method eventually returns (succeeds)

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 (succeeds)
- *Wait-free:* every thread calling a method eventually returns (succeeds)

Progress Conditions

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

Java Memory Model

- Java programming language does not guarantee linearizability when reading and writing fields of shared objects
- Due to compiler optimization memory reads and writes are often reordered

Singleton object

```
public static Singleton getInstance() {
  if (instance == null)
    instance = new Singleton();
  return instance;
}
```

Problem

Singleton object

- Create a single instance of the class
- Method must guard against multiple threads each seeing instance to be null and create new instances

Singleton object

```
public static Singleton getInstance() {
  synchronized(this) {
     if (instance == null)
          instance = new Singleton();
  }
     return instance;
}
```

Lock down critical section to avoid collisions

But what about optimization?

Singleton object

- Once the instance has been created, however no further synchronization should be necessary

Singleton object

```
public static Singleton getInstance() {
  if (instance == null) {
     synchronized(this) {
          instance = new Singleton();
     }
  }
  return instance;
}
```

What if two threads call synchronized simultaneously?

Singleton object

```
public static Singleton getInstance() {
  if (instance == null) {
     synchronized(this) {
          if (instance == null)
               instance = new Singleton();
     }
  }
  return instance;
}
```

Double-checked locking

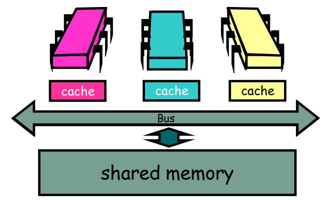
Singleton object

- In theory a double-checked lock is correct, however:
  - In theory, the constructor call takes place before the instance field is assigned
  - However, the java memory model allows these steps to occur out of order = making a partially initialized Singleton object visible to other programs

Java Memory Model

- In the Java memory model:
  - Objects reside in shared memory
  - Each thread has a private working memory that contains cached copies of fields it has read or written

Java Memory Model



Java Memory Model

- In the absence of explicit synchronization:
  - A thread that writes to a field may not update the memory right away, and
  - A thread that reads from a field may not update its working memory if the field's value in memory changes

Java Memory Model

- Need a way to force a thread to change the shared memory object when changing his own private memory copy and to read an object from the shared memory instead of reading from his private memory

Synchronization events

- Synchronization usually implies mutual exclusion
- In Java, is also implies reconciling a thread's working memory with the shared memory

Synchronization events

- In Java usually in one of two ways:
  - Cause a thread to write changes back to shared memory immediately
  - Cause thread to invalidate its working memory values and forces it to reread the fields from shared memory

Synchronization events

- Synchronization events are linearizable:
  - They are ordered
  - All threads agree on the ordering

Synchronization events

- Locks and synchronized blocks
- Volatile fields
- Final fields

Locks and synchronized blocks

- Thread can achieve mutual exclusion through implicit lock (synchronized block) or explicit lock
- If all accesses to a particular field are protected by the same lock, then the reads-writes to that field is linearizable

Locks and synchronized blocks

- When a thread releases a lock the changes are written to shared memory immediately
- When a thread acquires a lock it invalidates its own memory and rereads the value from shared memory

Volatile fields

- Reading a volatile field is like acquiring a lock – value is reread from shared memory
- Writing a volatile field is like releasing a lock – changes immediately written to shared memory

Volatile fields

- However, multiple reads-writes are not successful
- Some form of mutual exclusion is then needed

Final fields

- A field declared to be final cannot be modified once it has been initialized in its constructor

Final fields

```
class FinalFieldExample {
  final int x; int y;
  static FinalFieldExample;
  public FinalFieldExample() {
      x = 3;
      y = 4;
  }
  static void reader() {
      int i = x; int j = y;
  }
}
```

Correct!
Thread that calls reader() is guaranteed to see x equal to 3

In summary

- Reads-writes to fields are linearizable if:
  - The field is volatile
  - The field is protected by a unique lock used by all readers and writers