Parallel Programming
Exercise Session 12

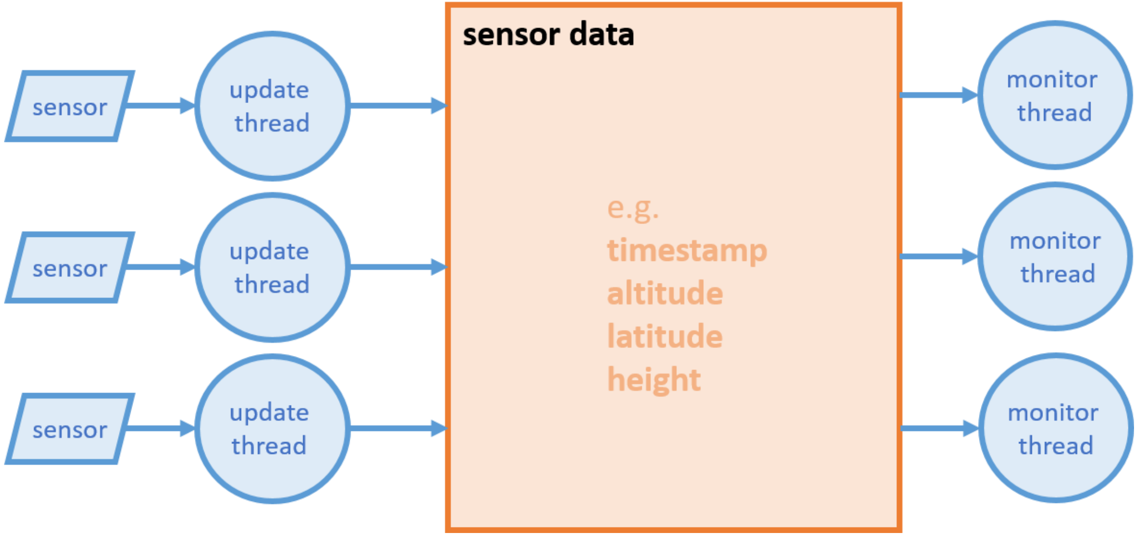
Outline

1. Feedback: Assignment 11
2. Recap: Linearizability
3. Recap: Java memory model
4. Recap: Software Transactional Memory (STM)
5. Assignment 12

Feedback: Assignment 11

Assignment 11

- Multisensor System.



LockedSensors

```
class LockedSensors implements Sensors {

    long time = 0;
    double data[];

    private ReadWriteLock lock;
    private Lock readlock;
    private Lock writelock;

    LockedSensors() {
        this(new ReadWriteMonitorLock());
    }

    LockedSensors(ReadWriteLock l){
        time = 0;
        lock = l;
        readlock = lock.readLock();
        writelock = lock.writeLock();
    }
```

LockedSensors

```
class LockedSensors implements Sensors {

    long time = 0;
    double data[];

    private ReadWriteLock lock;
    private Lock readlock;
    private Lock writelock;

    LockedSensors() {
        this(new ReadWriteMonitorLock());
    }

    LockedSensors(ReadWriteLock l){
        time = 0;
        lock = l;
        readlock = lock.readLock();
        writelock = lock.writeLock();
    }
```

```
public long get(double val[])
{


        if (time == 0)
            return 0;
        else{
            for (int i = 0; i<data.length; ++i)
                val[i] = data[i];
            return time;
        }



}

public void update(long timestamp, double[] data)
{


        if (timestamp > time) {
            if (this.data == null)
                this.data = new double[data.length];
            time = timestamp;
            for (int i=0; i<data.length;++i)
                this.data[i]= data[i];
            }



}
```

LockedSensors

```
class LockedSensors implements Sensors {

    long time = 0;
    double data[];

    private ReadWriteLock lock;
    private Lock readlock;
    private Lock writelock;

    LockedSensors() {
        this(new ReadWriteMonitorLock());
    }

    LockedSensors(ReadWriteLock l){
        time = 0;
        lock = l;
        readlock = lock.readLock();
        writelock = lock.writeLock();
    }
```

```
public long get(double val[])
{
    readlock.lock();
    try{
        if (time == 0)
            return 0;
        else{
            for (int i = 0; i<data.length; ++i)
                val[i] = data[i];
            return time;
        }
    }finally {
        readlock.unlock();
    }

}

public void update(long timestamp, double[] data)
{
    writelock.lock();
    try{
        if (timestamp > time) {
            if (this.data == null)
                this.data = new double[data.length];
            time = timestamp;
            for (int i=0; i<data.length;++i)
                this.data[i]= data[i];
            }
    }
    finally {
        writelock.unlock();
    }
}
```

Lock implementation

```
public class ReadWriteMonitorLock implements ReadWriteLock{
    private Lock readerlock = new ReadMonitorLock(this);
    private Lock writerlock = new WriteMonitorLock(this);

    //Invariant 0<=readers /\ 0<=writers<=1 /\ readers*writers=0
    private int readers=0;
    private int writers=0;

    private int writersWating=0;
    private int readersWating=0;
    private int readersToWait=0;

    @Override
    public Lock readLock() {
        return readerlock;
    }

    @Override
    public Lock writeLock() {
        return writerlock;
    }
```

Lock implementation

```
public class ReadWriteMonitorLock implements ReadWriteLock{
    private Lock readerlock = new ReadMonitorLock(this);
    private Lock writerlock = new WriteMonitorLock(this);

    //Invariant 0<=readers /\ 0<=writers<=1 /\ readers*writers=0
    private int readers=0;
    private int writers=0;

    private int writersWating=0;
    private int readersWating=0;
    private int readersToWait=0;

    @Override
    public Lock readLock() {
        return readerlock;
    }

    @Override
    public Lock writeLock() {
        return writerlock;
    }
```

```
private synchronized void aquireRead(){
    readersWating++;
    while(                                      ){
        try {
            wait();
        } catch (InterruptedException e) { e.printStackTrace(); }
    }
    readersWating--;
    readersToWait--;
    readers++;
}
private synchronized void releaseRead(){
    readers--;
    notifyAll();
}
```

Lock implementation

```
public class ReadWriteMonitorLock implements ReadWriteLock{
    private Lock readerlock = new ReadMonitorLock(this);
    private Lock writerlock = new WriteMonitorLock(this);

    //Invariant 0<=readers /\ 0<=writers<=1 /\ readers*writers=0
    private int readers=0;
    private int writers=0;

    private int writersWating=0;
    private int readersWating=0;
    private int readersToWait=0;

    @Override
    public Lock readLock() {
        return readerlock;
    }

    @Override
    public Lock writeLock() {
        return writerlock;
    }
```

```
private synchronized void aquireRead(){
    readersWating++;
    while(writers>0 || (writersWating>0 && readersToWait<=0)){
        try {
            wait();
        } catch (InterruptedException e) { e.printStackTrace(); }
    }
    readersWating--;
    readersToWait--;
    readers++;
}
private synchronized void releaseRead(){
    readers--;
    notifyAll();
}
```

Lock implementation

```java
public class ReadWriteMonitorLock implements ReadWriteLock{
    private Lock readerlock = new ReadMonitorLock(this);
    private Lock writerlock = new WriteMonitorLock(this);

    //Invariant 0<=readers /\ 0<=writers<=1 /\ readers*writers=0
    private int readers=0;
    private int writers=0;

    private int writersWating=0;
    private int readersWating=0;
    private int readersToWait=0;

    @Override
    public Lock readLock() {
        return readerlock;
    }

    @Override
    public Lock writeLock() {
        return writerlock;
    }
```

```java
private synchronized void aquireRead(){
    readersWating++;
    while(writers>0 || (writersWating>0 && readersToWait<=0)){
        try {
            wait();
        } catch (InterruptedException e) { e.printStackTrace(); }
    }
    readersWating--;
    readersToWait--;
    readers++;
}
private synchronized void releaseRead(){
    readers--;
    notifyAll();
}
```

```java
private synchronized void aquireWrite(){
    writersWating++;
    while(                                        ){
        try {
            wait();
        } catch (InterruptedException e) { e.printStackTrace(); }
    }
    writersWating--;
    writers++;
}
private synchronized void releaseWrite(){
    writers--;
    readersToWait = readersWating;
    notifyAll();
}
```

Lock implementation

```
public class ReadWriteMonitorLock implements ReadWriteLock{
    private Lock readerlock = new ReadMonitorLock(this);
    private Lock writerlock = new WriteMonitorLock(this);

    //Invariant 0<=readers /\ 0<=writers<=1 /\ readers*writers=0
    private int readers=0;
    private int writers=0;

    private int writersWating=0;
    private int readersWating=0;
    private int readersToWait=0;

    @Override
    public Lock readLock() {
        return readerlock;
    }

    @Override
    public Lock writeLock() {
        return writerlock;
    }
```

```
private synchronized void aquireRead(){
    readersWating++;
    while(writers>0 || (writersWating>0 && readersToWait<=0)){
        try {
            wait();
        } catch (InterruptedException e) { e.printStackTrace(); }
    }
    readersWating--;
    readersToWait--;
    readers++;
}
private synchronized void releaseRead(){
    readers--;
    notifyAll();
}


private synchronized void aquireWrite(){
    writersWating++;
    while(writers>0 || readers>0 || readersToWait>0){
        try {
            wait();
        } catch (InterruptedException e) { e.printStackTrace(); }
    }
    writersWating--;
    writers++;
}
private synchronized void releaseWrite(){
    writers--;
    readersToWait = readersWating;
    notifyAll();
}
```

LockFreeSensors

```
class LockFreeSensors implements Sensors {

    AtomicReference<SensorData> data;

    LockFreeSensors()
    {
        data = new AtomicReference<SensorData>(new SensorData(0L, new double[0]));
    }
```

LockFreeSensors

```
class LockFreeSensors implements Sensors {

    AtomicReference<SensorData> data;

    LockFreeSensors()
    {
         data = new AtomicReference<SensorData>(new SensorData(0L, new double[0]));
    }
```

Lecture 20
Without Locks II

LockFreeSensors

```
class LockFreeSensors implements Sensors {

    AtomicReference<SensorData> data;

    LockFreeSensors()
    {
        data = new AtomicReference<SensorData>(new SensorData(0L, new double[0]));
    }
```

```
public long get(double val[])
{
    SensorData d = data.get();
    double[] v = d.getValues();
    if (v == null) return 0;
    for (int i=0; i<v.length; ++i)
        val[i] = v[i];
    return d.getTimestamp();
}
```

LockFreeSensors

```
class LockFreeSensors implements Sensors {

    AtomicReference<SensorData> data;

    LockFreeSensors()
    {
        data = new AtomicReference<SensorData>(new SensorData(0L, new double[0]));
    }
```

```
public long get(double val[])
{
    SensorData d = data.get();
    double[] v = d.getValues();
    if (v == null) return 0;
    for (int i=0; i<v.length; ++i)
        val[i] = v[i];
    return d.getTimestamp();
}
```

```
public void update(long timestamp, double[] val)
{
    SensorData old_data;
    SensorData new_data = new SensorData(timestamp, val);
    do {
        old_data = data.get();
        if (old_data != null && old_data.getTimestamp() >= new_data.getTimestamp()) {
            return;
        }
    } while (!data.compareAndSet(old_data, new_data));

}
```

LockFreeSensors

```
class LockFreeSensors implements Sensors {

    AtomicReference<SensorData> data;

    LockFreeSensors()
    {
        data = new AtomicReference<SensorData>(new SensorData(0L, new double[0]));
    }
```

```
public long get(double val[])
{
    SensorData d = data.get();
    double[] v = d.getValues();
    if (v == null) return 0;
    for (int i=0; i<v.length; ++i)
        val[i] = v[i];
    return d.getTimestamp();
}
```

```
public void update(long timestamp, double[] val)
{
    SensorData old_data;
    SensorData new_data = new SensorData(timestamp, val);
    do {
        old_data = data.get();
        if (old_data != null && old_data.getTimestamp() >= new_data.getTimestamp()) {
            return;
        }
    } while (!data.compareAndSet(old_data, new_data));

}
```

If vs while !

Correctness

Program correctness in a sequential world

Objects encapsulate some representation of state

Program correctness in a sequential world

Objects encapsulate some representation of state

- We don’t reason about the representation directly, but about its visibility from outside (via public methods) (e.g. stack.top()==3)

Program correctness in a sequential world

Objects encapsulate some representation of state

- We don’t reason about the representation directly, but about its visibility from outside (via public methods) (e.g. stack.top()==3)
- State must be consistent, i.e., according to the public class invariant (e.g., forall x. stack.push(x).pop()==x)

Program correctness in a sequential world

Objects encapsulate some representation of state

- We don’t reason about the representation directly, but about its visibility from outside (via public methods) (e.g. stack.top()==3)
- State must be consistent, i.e., according to the public class invariant (e.g., forall x. stack.push(x).pop()==x)
- Each method satisfies its post-condition, given its pre-condition

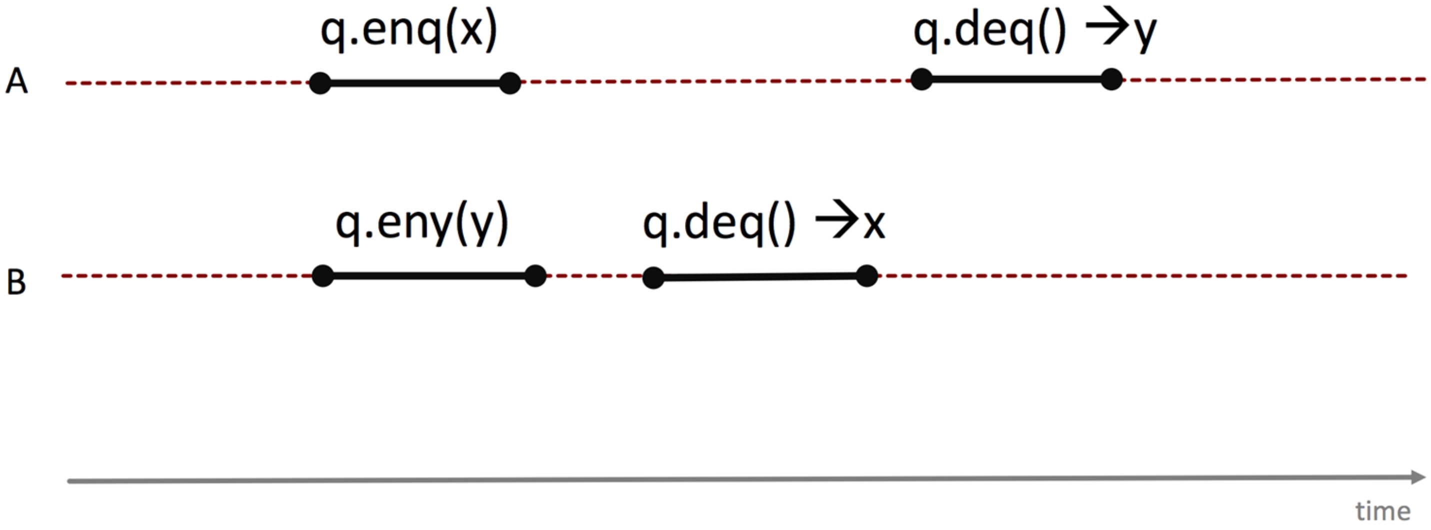
Program correctness in a concurrent world

Sequential
Each method described independently.
Object's state is defined between method calls.
Adding new method does not affect older methods.

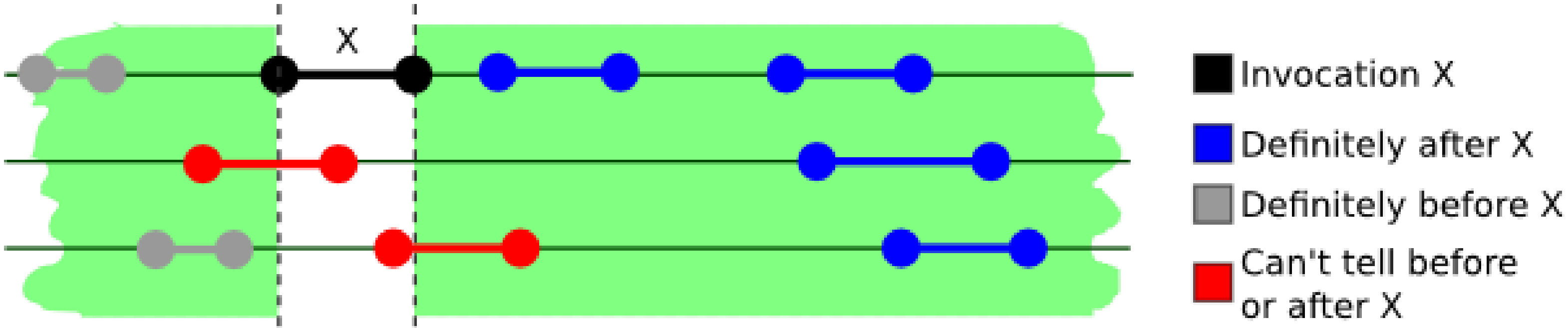
Program correctness in a concurrent world

Sequential	Concurrent
Each method described independently.	Need to describe all possible interactions between methods. (what if enq and deq overlap? …)
Object's state is defined between method calls.	Because methods can overlap, the object may never be between method calls…
Adding new method does not affect older methods.	Need to think about all possible interactions with the new method.

Execution



Execution


X
Invocation X
Definitely after X
Definitely before X
Can't tell before
or after X

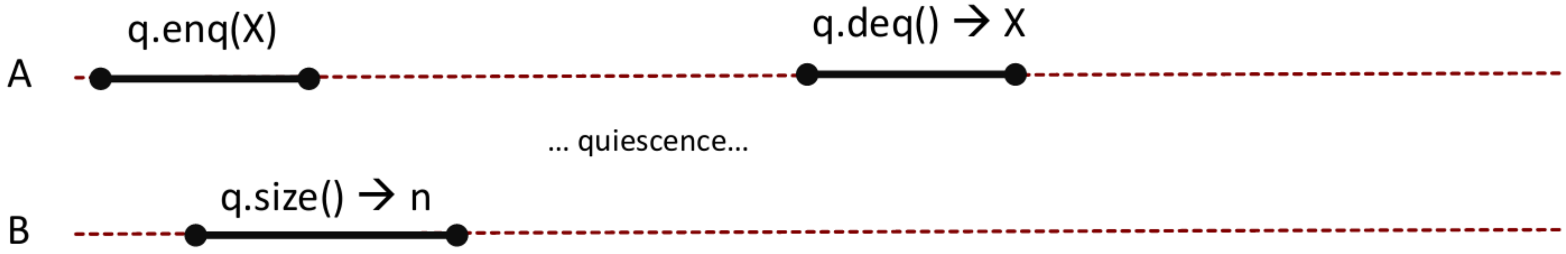
Quiescent Consistency

Quiescent consistency

requires non-overlapping operations to appear to take effect in their real-time order, but overlapping operations might be reordered

Quiescent consistency

requires non-overlapping operations to appear to take effect in their real-time order, but overlapping operations might be reordered



Sequential Consistency

Sequential consistency

A multiprocessing system has sequential consistency if:

"...the results of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."

- Leslie Lamport (inventor of sequential consistency)

Sequential consistency requirements

1. All instructions are executed in order.

Sequential consistency requirements

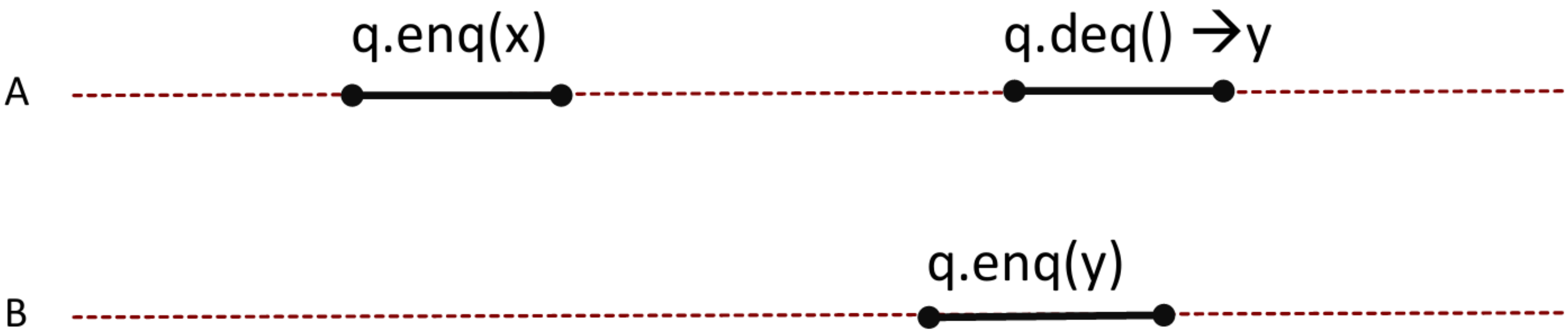
1. All instructions are executed in order.

2. Every write operation becomes instantaneously visible throughout the system.

Sequential consistency requirements

1. All instructions are executed in order.

2. Every write operation becomes instantaneously visible throughout the system.



Sequential consistency vs Quiescent consistency

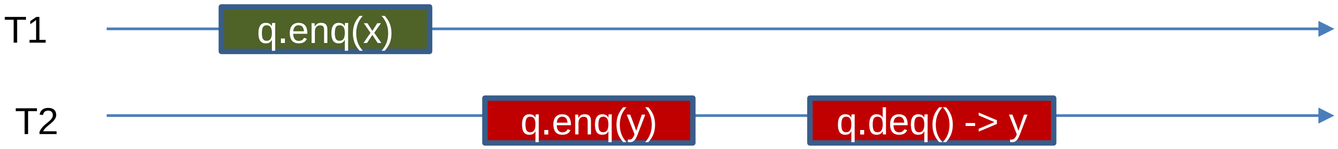
sequential consistency and quiescent consistency are incomparable:

there exist sequentially consistent executions that are not
quiescently consistent, and vice versa

Sequential consistency vs Quiescent consistency

sequential consistency and quiescent consistency are <u>incomparable</u>:

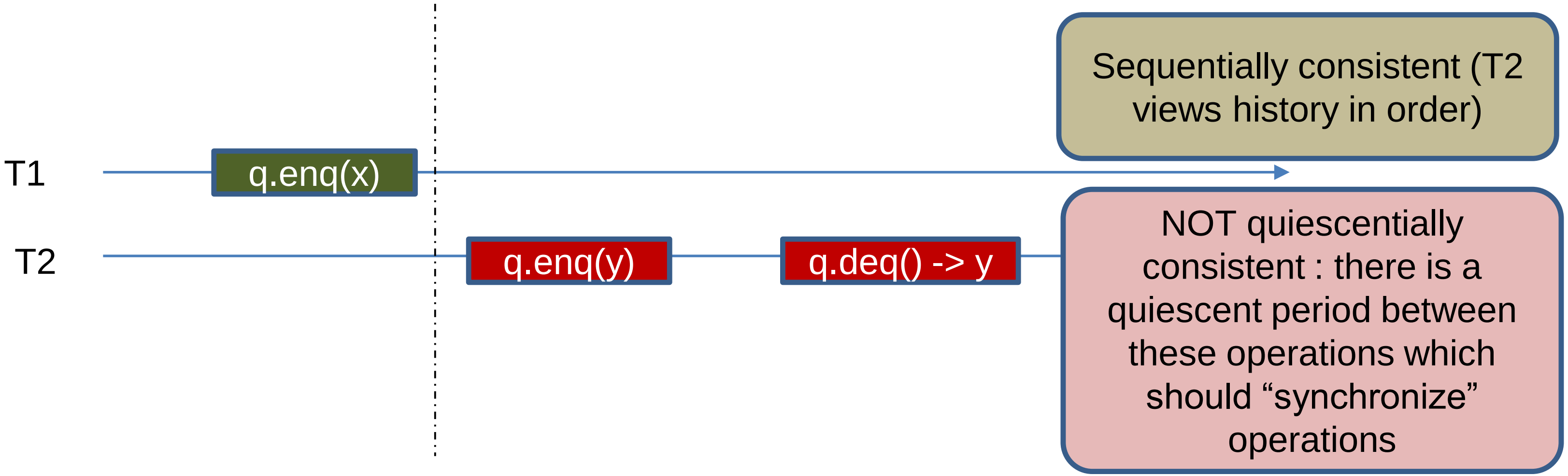
there exist sequentially consistent executions that are not
quiescently consistent, and vice versa



Sequential consistency vs Quiescent consistency

sequential consistency and quiescent consistency are <u>incomparable</u>:

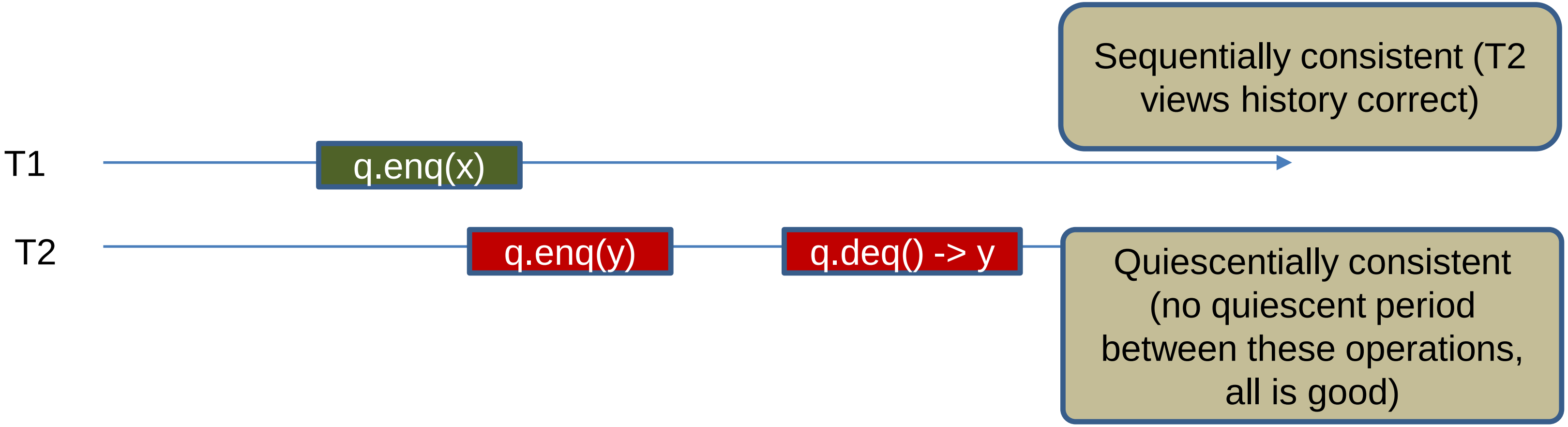
there exist sequentially consistent executions that are not quiescently consistent, and vice versa



Sequential consistency vs Quiescent consistency

sequential consistency and quiescent consistency are incomparable:

there exist sequentially consistent executions that are not quiescently consistent, and vice versa



Sequential consistency vs Quiescent consistency

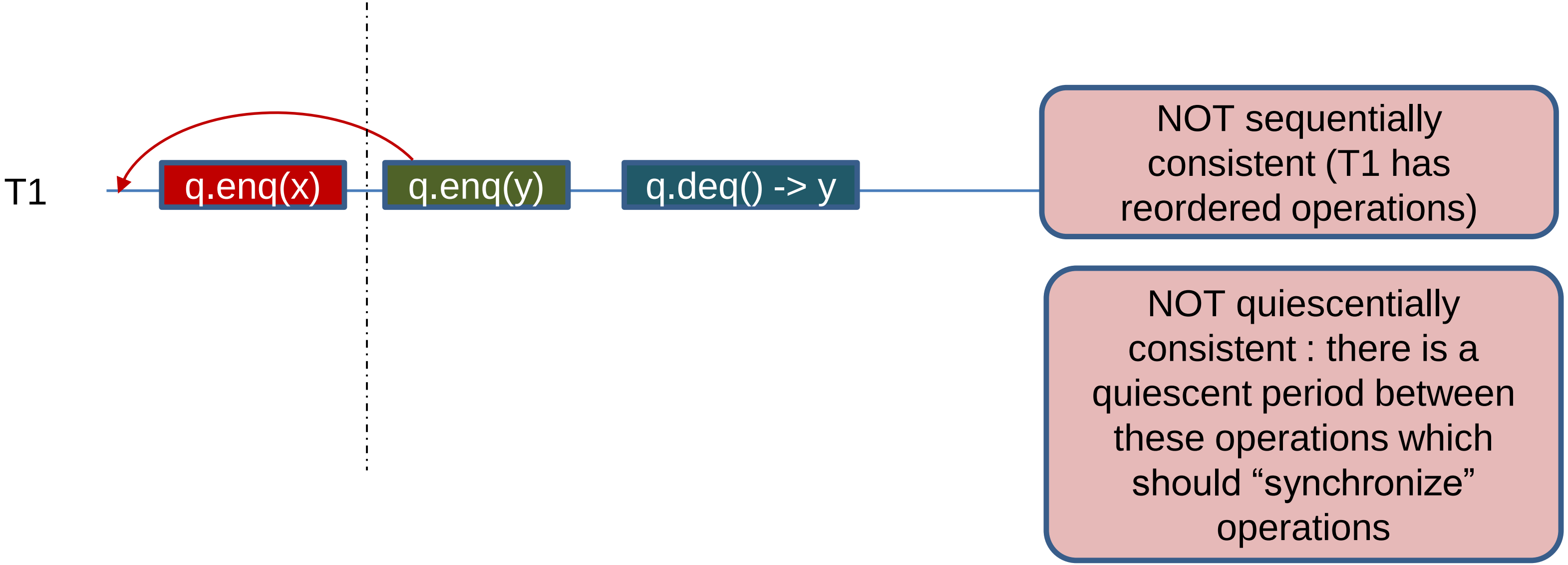
sequential consistency and quiescent consistency are <u>incomparable</u>:

there exist sequentially consistent executions that are not quiescently consistent, and vice versa

Sequential consistency vs Quiescent consistency

sequential consistency and quiescent consistency are <u>incomparable</u>:

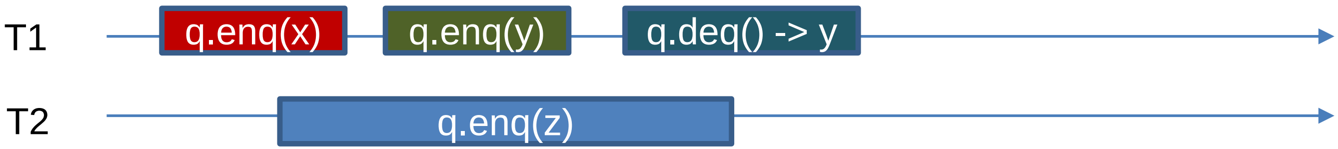
there exist sequentially consistent executions that are not quiescently consistent, and vice versa



Sequential consistency vs Quiescent consistency

sequential consistency and quiescent consistency are <u>incomparable</u>:

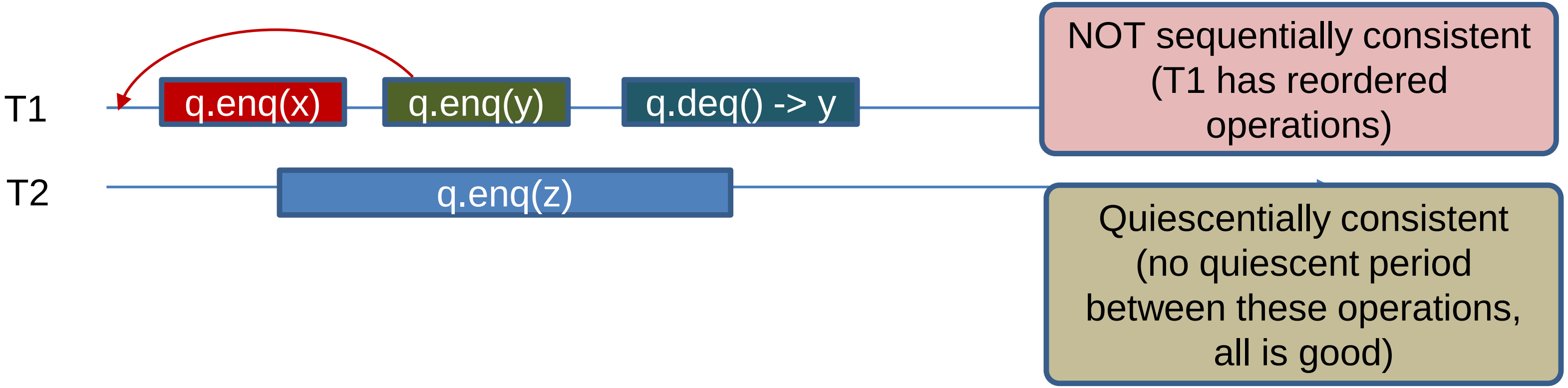
there exist sequentially consistent executions that are not quiescently consistent, and vice versa



Sequential consistency vs Quiescent consistency

sequential consistency and quiescent consistency are incomparable:

there exist sequentially consistent executions that are not quiescently consistent, and vice versa



Linearizability

Consistency model: Linearizability

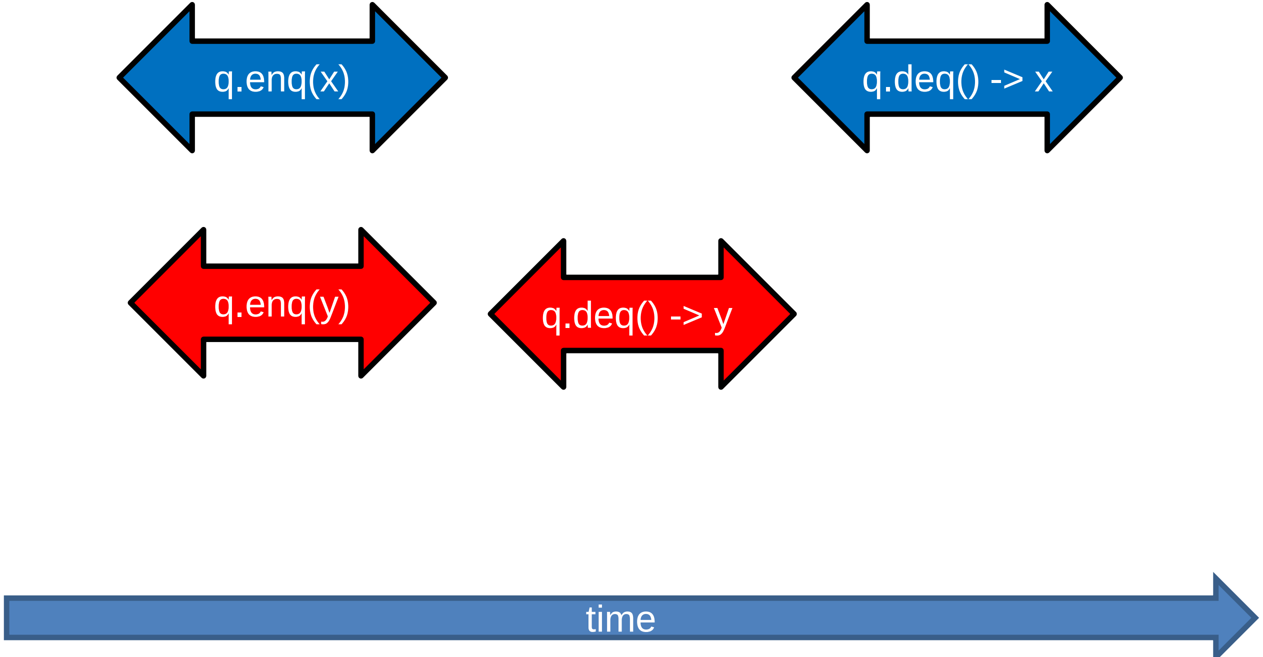
- Linearizability provides the illusion that each operation applied by concurrent processes takes effect instantaneously between its invocation and its response.

Consistency model: Linearizability

- Linearizability provides the illusion that each operation applied by concurrent processes takes effect instantaneously between its invocation and its response.
- An object for which this is true for all possible executions is called **linearizable**

Example (1)

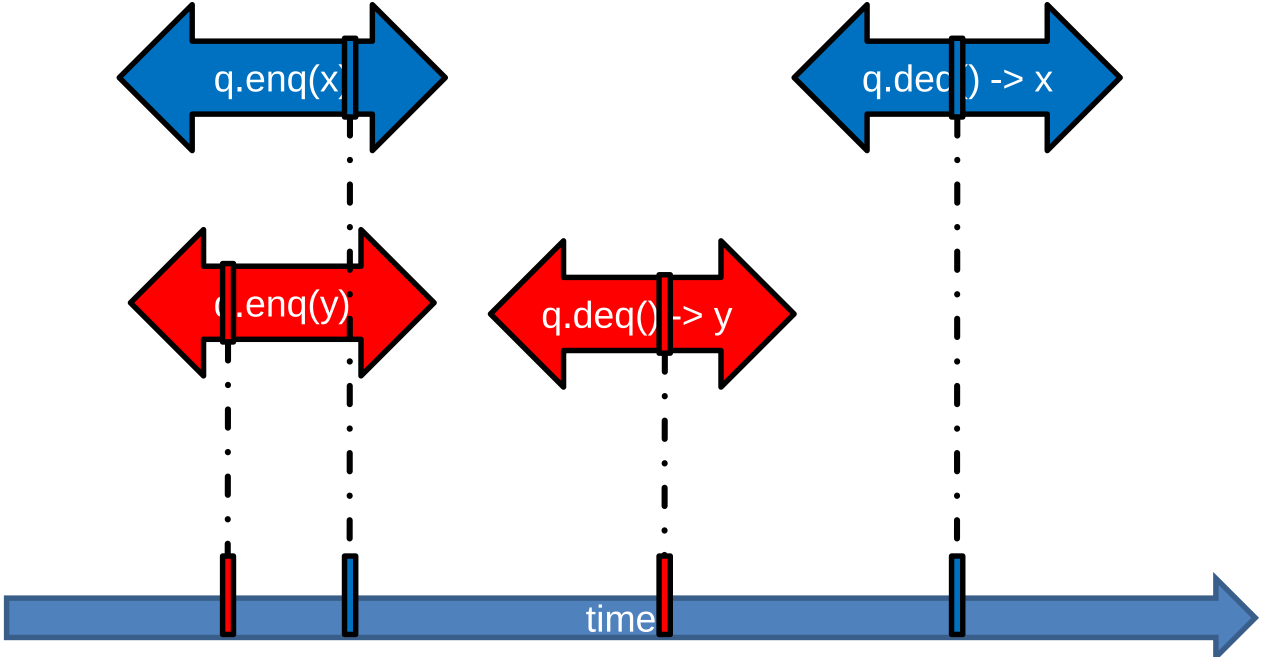
Is this linearizable?



Example (1)

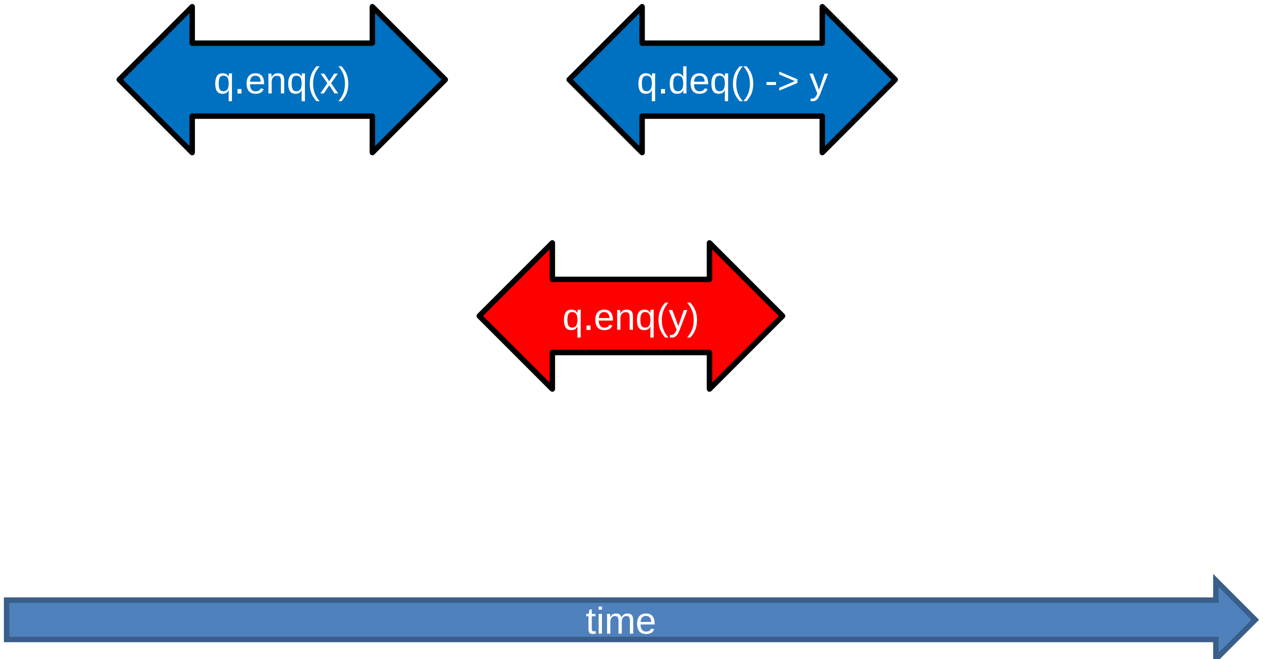
Is this linearizable?

Yes!



Example (2)

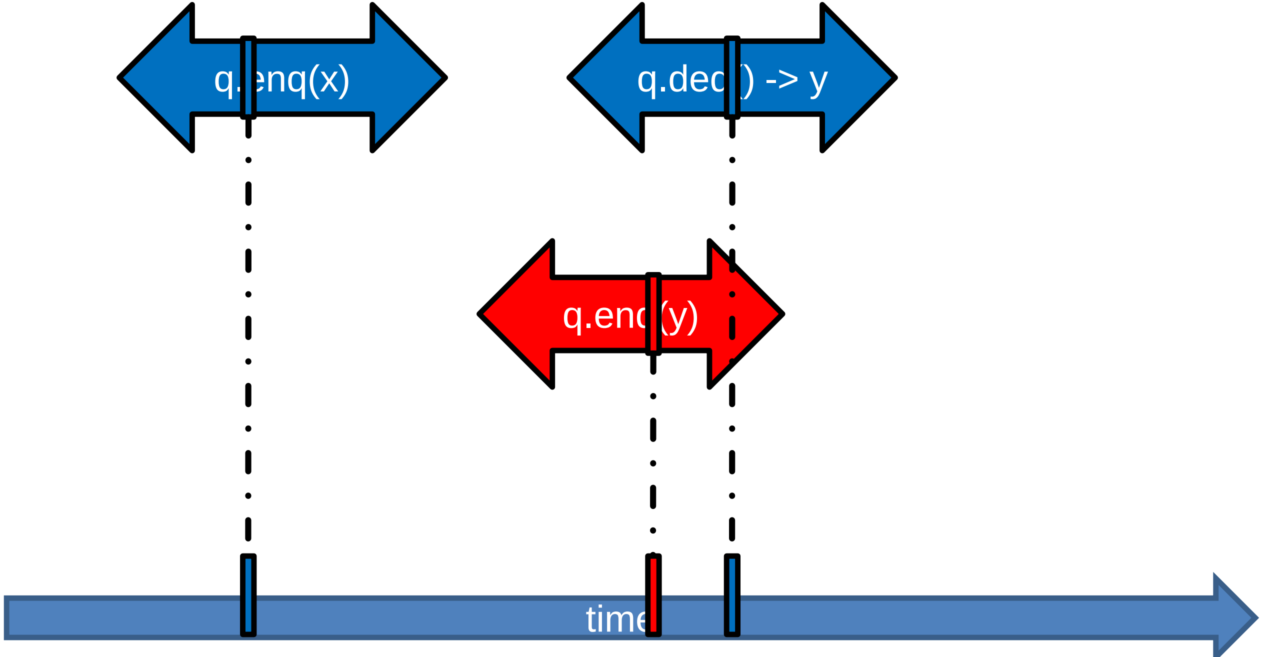
Is this linearizable?



Example (2)

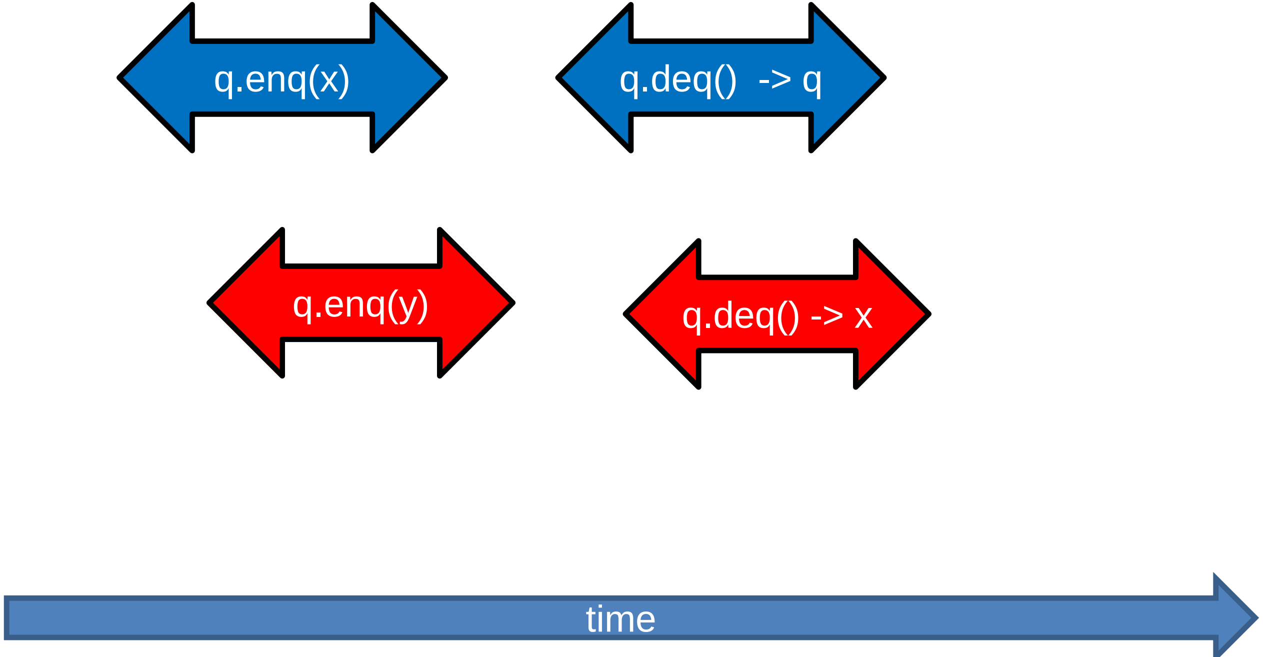
Is this linearizable?

No!



Example (3)

Is this linearizable?

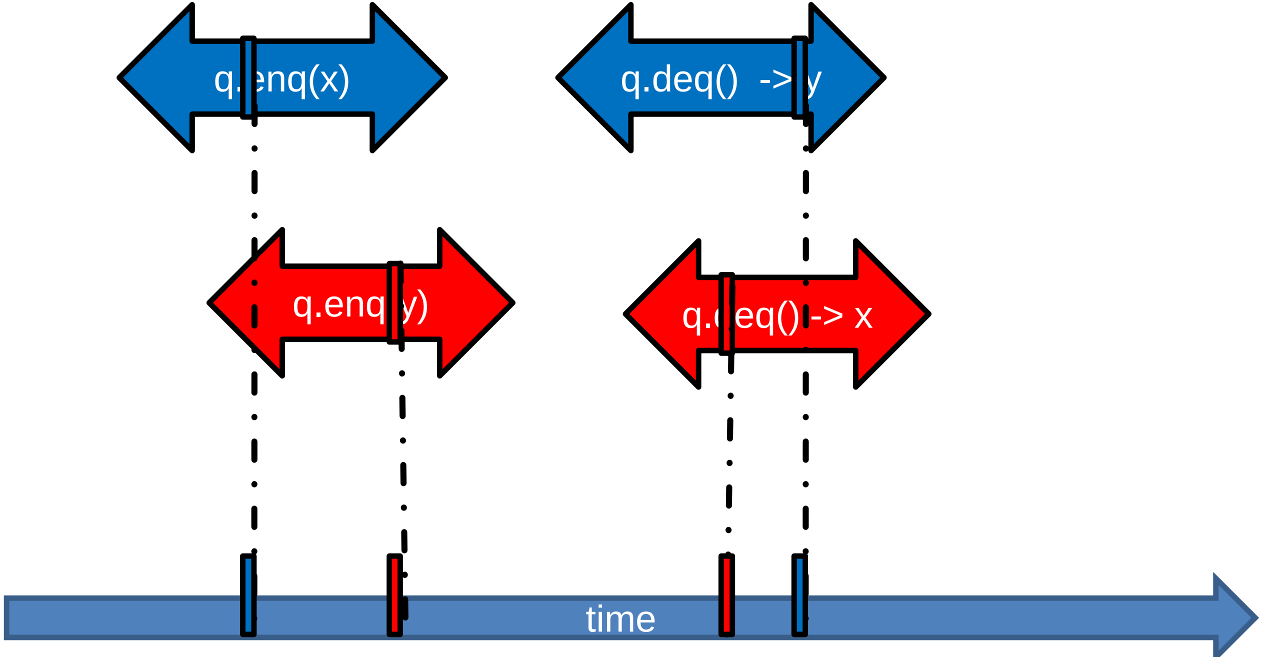


Example (3)

Here we got multiple orders!

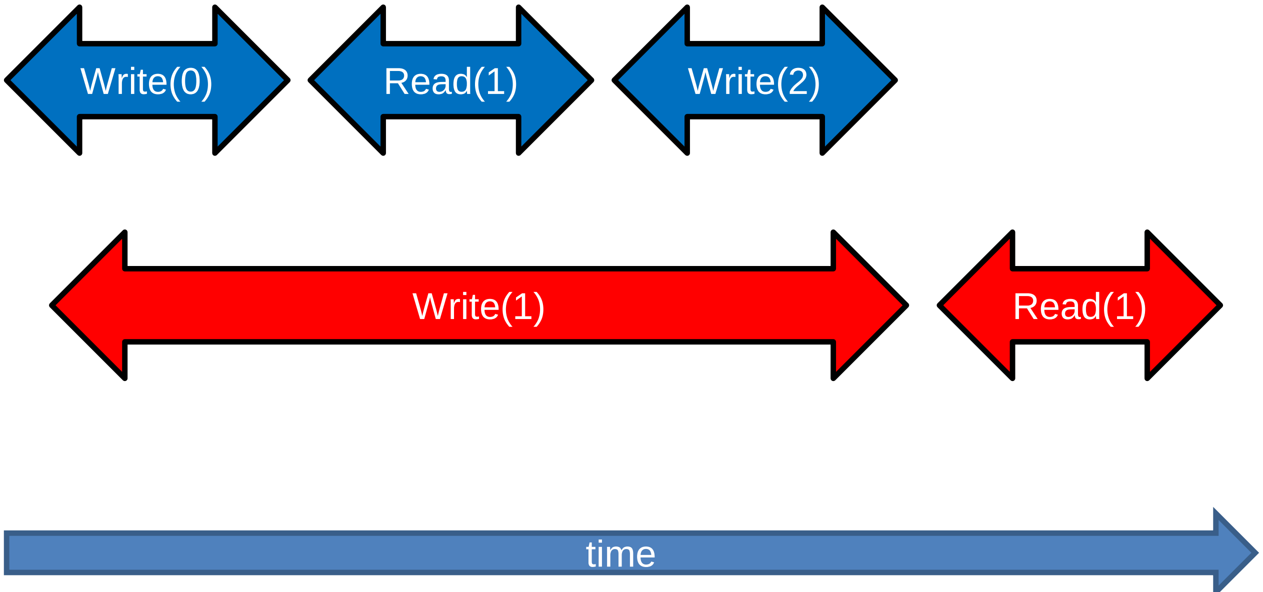
Is this linearizable?

Yes!



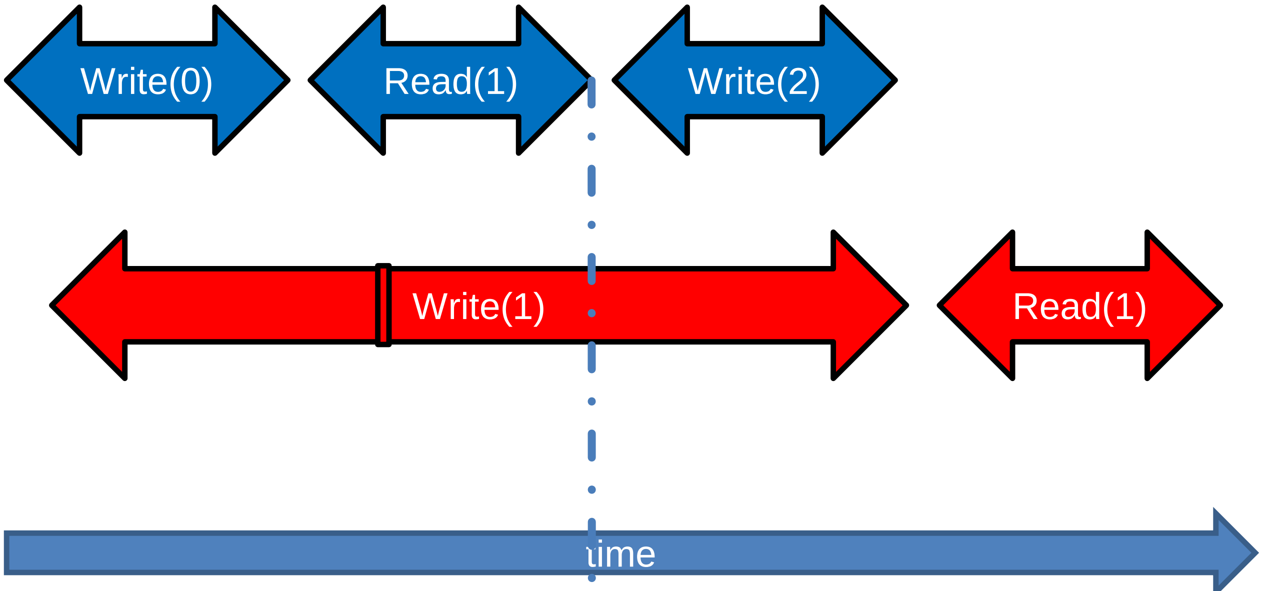
Example (4)

Is this linearizable?



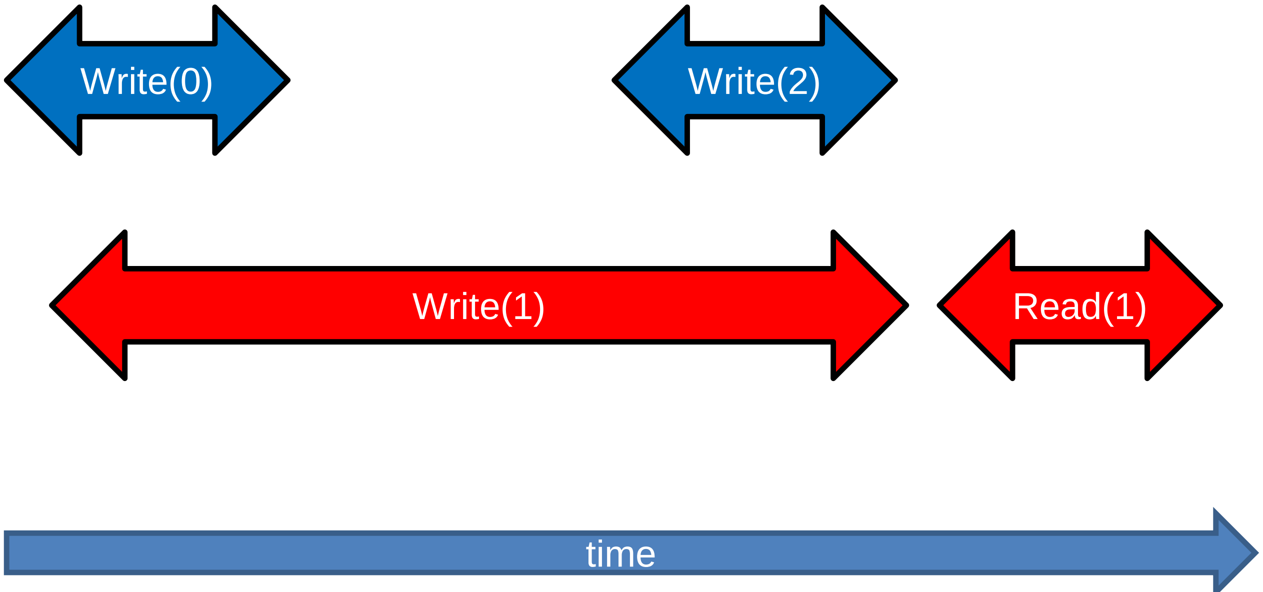
Example (4)

Is this linearizable? **No!**



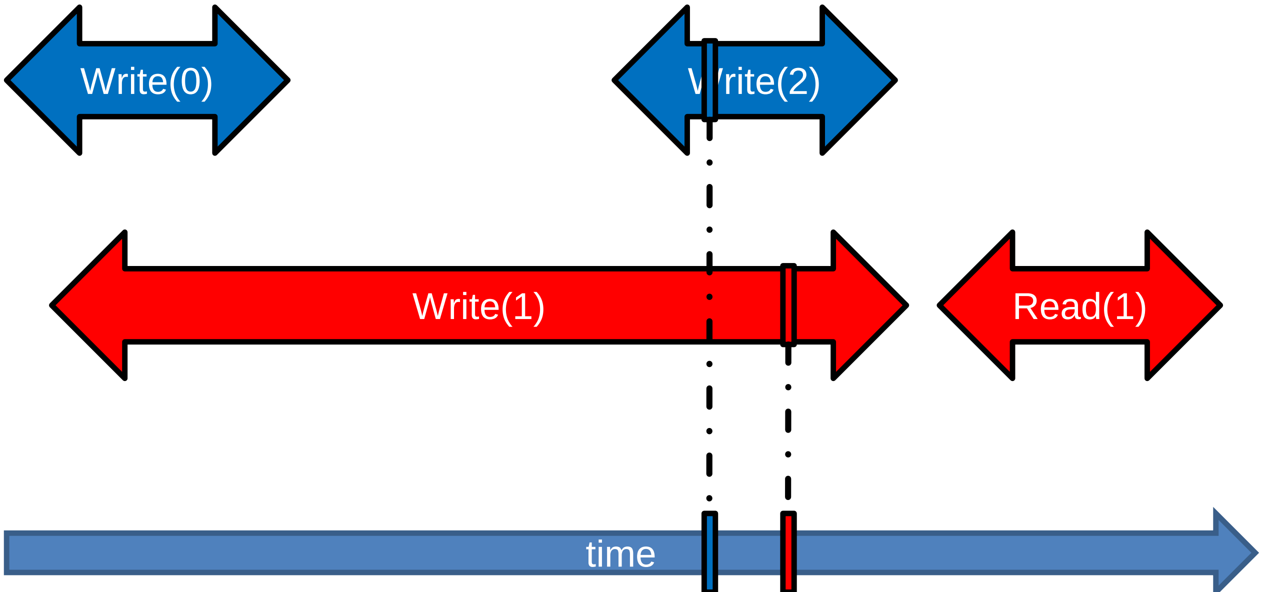
Example (4.5)

Is this linearizable?



Example (4.5)

Is this linearizable? **Yes!**



Linearization Point

```
public boolean add(T item) {
  int key = item.hashCode();
  head.lock();
  Node pred = head;
  try {
    Node curr = pred.next;
    curr.lock();
    try {
      while (curr.key < key) {
        pred.unlock();
        pred = curr;
        curr = curr.next;
        curr.lock();
      }
      if (curr.key == key) {
        return false;
      }
      Node newNode = new Node(item);
      newNode.next = curr;
      pred.next = newNode;
      return true;
    } finally {
      curr.unlock();
    }
  } finally {
    pred.unlock();
  }
}
```

The linearization point is the point where the method takes effect.

Linearization Point

```
public boolean add(T item) {
  int key = item.hashCode();
  head.lock();
  Node pred = head;
  try {
    Node curr = pred.next;
    curr.lock();
    try {
      while (curr.key < key) {
        pred.unlock();
        pred = curr;
        curr = curr.next;
        curr.lock();
      }
      if (curr.key == key) {
        return false;
      }
      Node newNode = new Node(item);
      newNode.next = curr;
      pred.next = newNode;
      return true;
    } finally {
      curr.unlock();
    }
  } finally {
    pred.unlock();
  }
}
```

The linearization point is the point where the method takes effect.

Linearization Point

```
class WaitFreeQueue<T> {
  volatile int head = 0, tail = 0;
  T[] items;
  public WaitFreeQueue(int capacity) {
    items = (T[])new Object[capacity];
    head = 0; tail = 0;
  }
  public void enq(T x) throws FullException {
    if (tail - head == items.length)
      throw new FullException();
    items[tail % items.length] = x;
    tail++;
  }
  public T deq() throws EmptyException {
    if (tail - head == 0)
      throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  }
}
```

The linearization point is the point where the method takes effect.

Linearization Point

```
class WaitFreeQueue<T> {
  volatile int head = 0, tail = 0;
  T[] items;
  public WaitFreeQueue(int capacity) {
    items = (T[])new Object[capacity];
    head = 0; tail = 0;
  }
  public void enq(T x) throws FullException {
    if (tail - head == items.length)
      throw new FullException();
    items[tail % items.length] = x;
    tail++;
  }
  public T deq() throws EmptyException {
    if (tail - head == 0)
      throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  }
}
```

The linearization point is the point where the method takes effect.

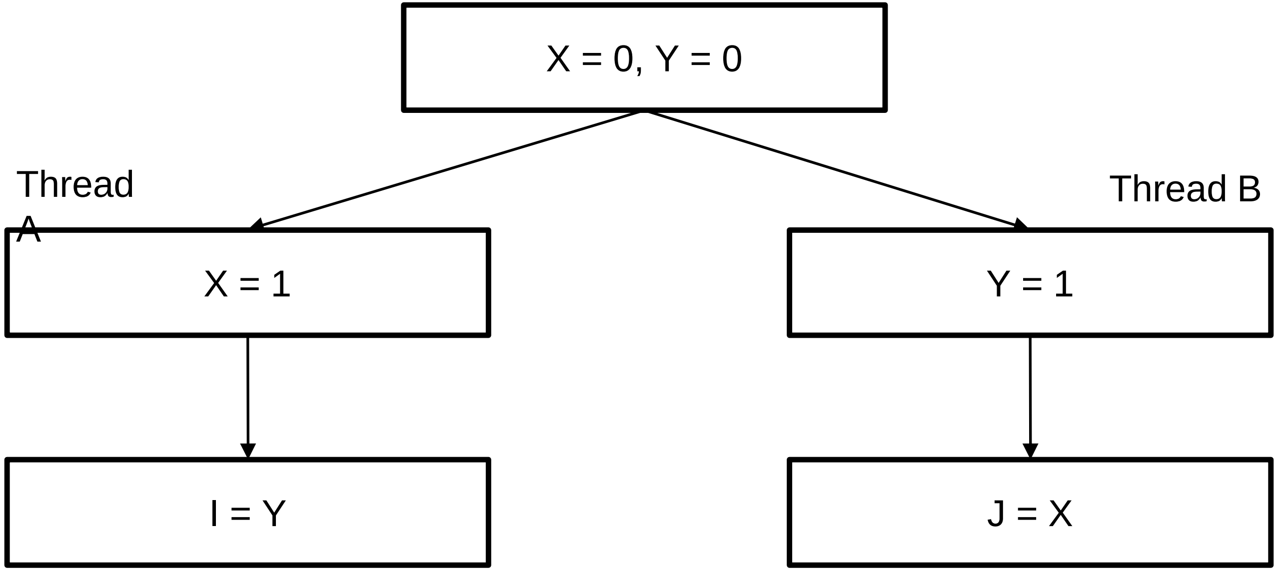
Linearization Point

```java
class WaitFreeQueue<T> {
  volatile int head = 0, tail = 0;
  T[] items;
  public WaitFreeQueue(int capacity) {
    items = (T[])new Object[capacity];
    head = 0; tail = 0;
  }
  public void enq(T x) throws FullException {
    if (tail - head == items.length)
      throw new FullException();
    items[tail % items.length] = x;
    tail++;
  }
  public T deq() throws EmptyException {
    if (tail - head == 0)
      throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  }
}
```

The linearization point is the point where the method takes effect.

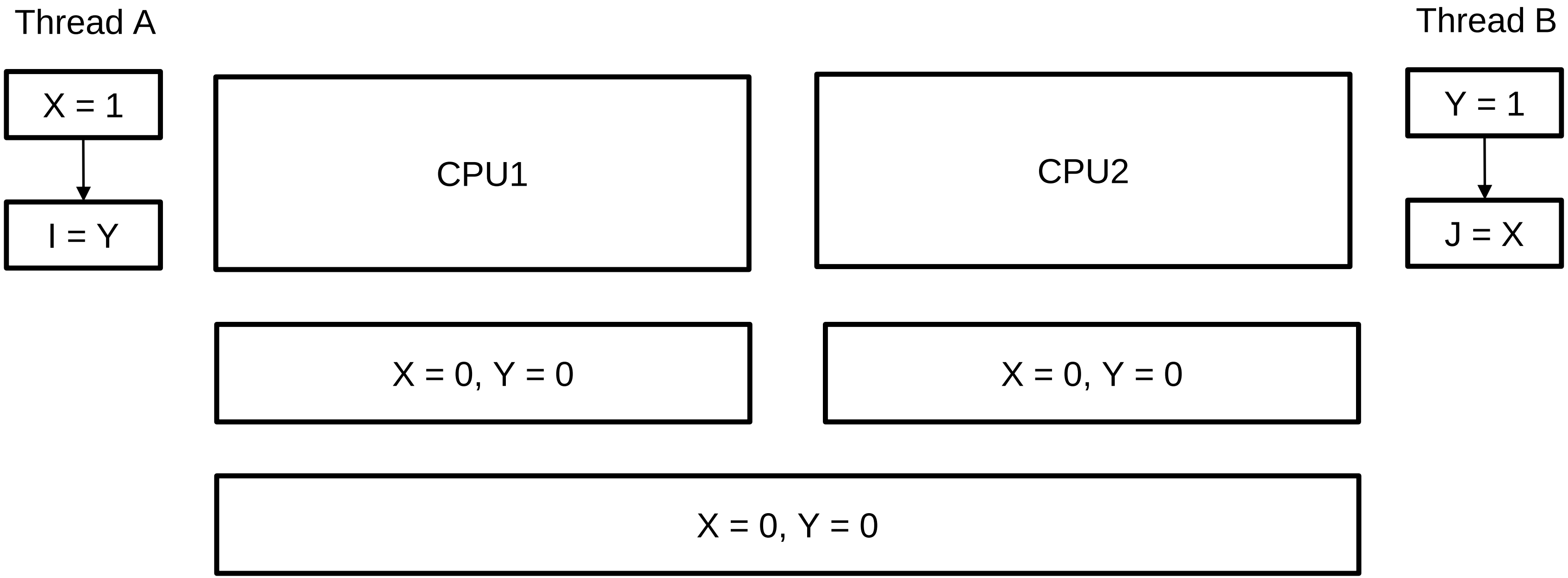
Recap: Java Memory Model

Quiz

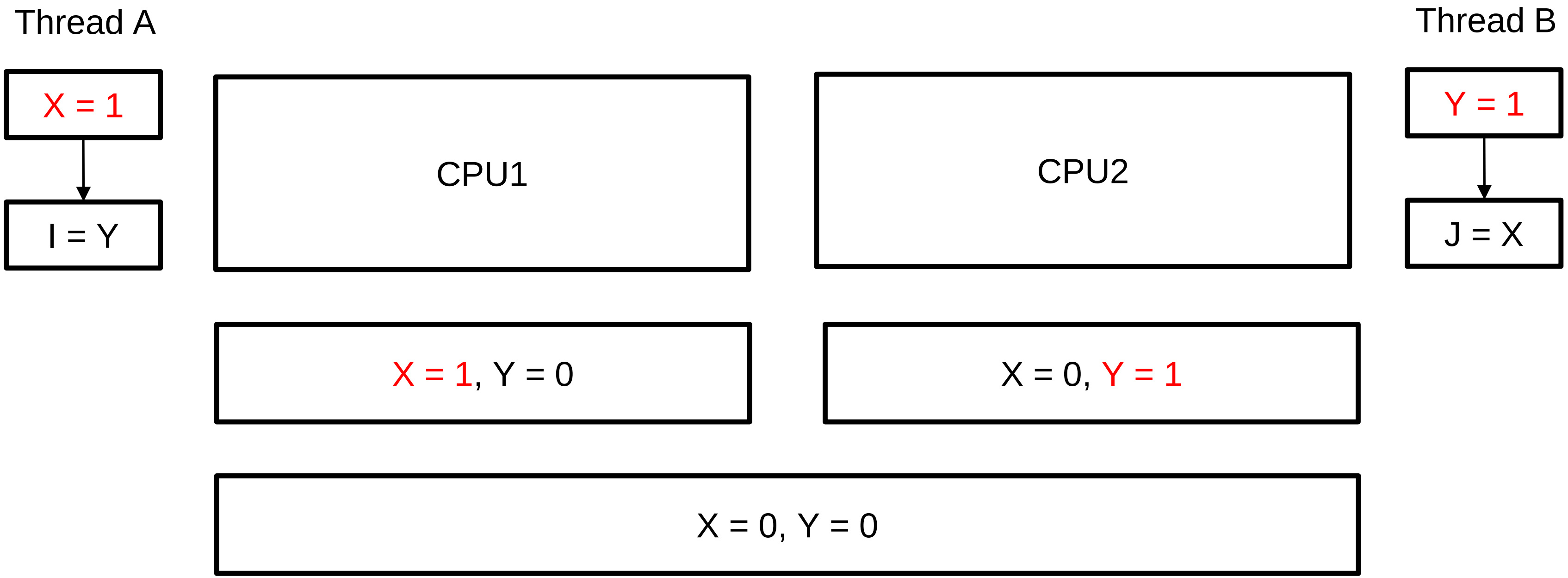


Can I == 0 and J == 0 at the end of the execution?

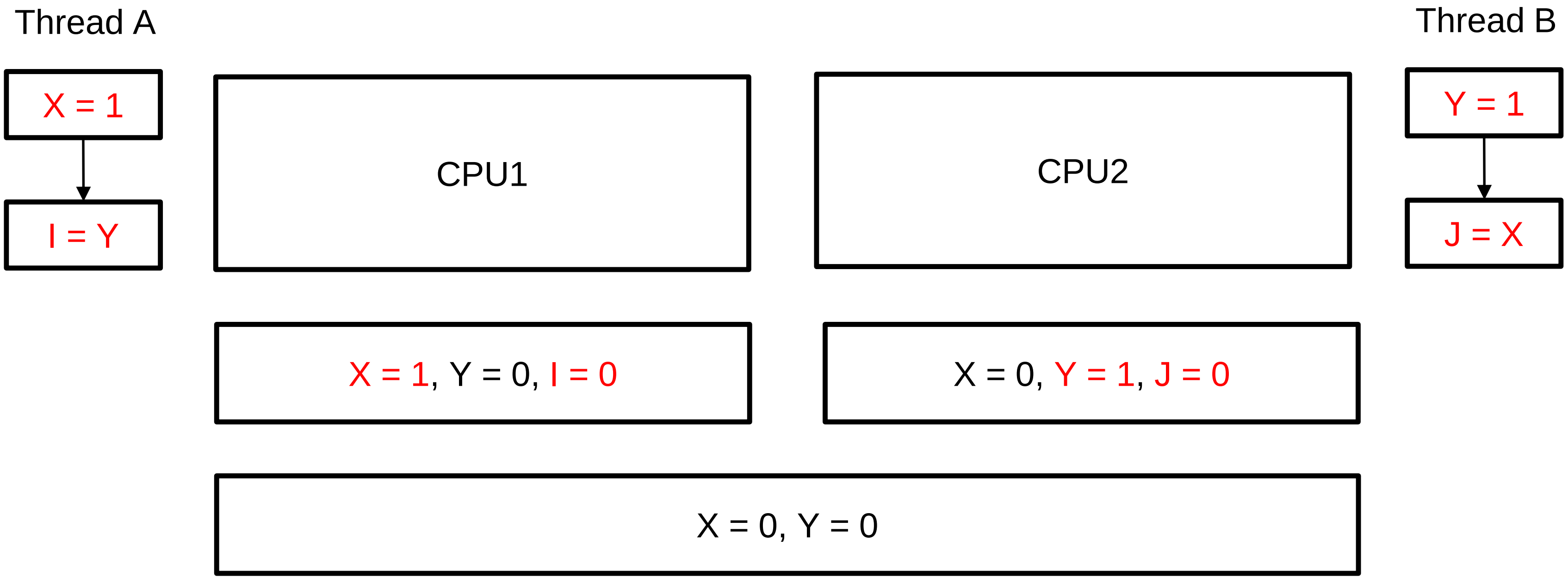
Quiz – Multicore case



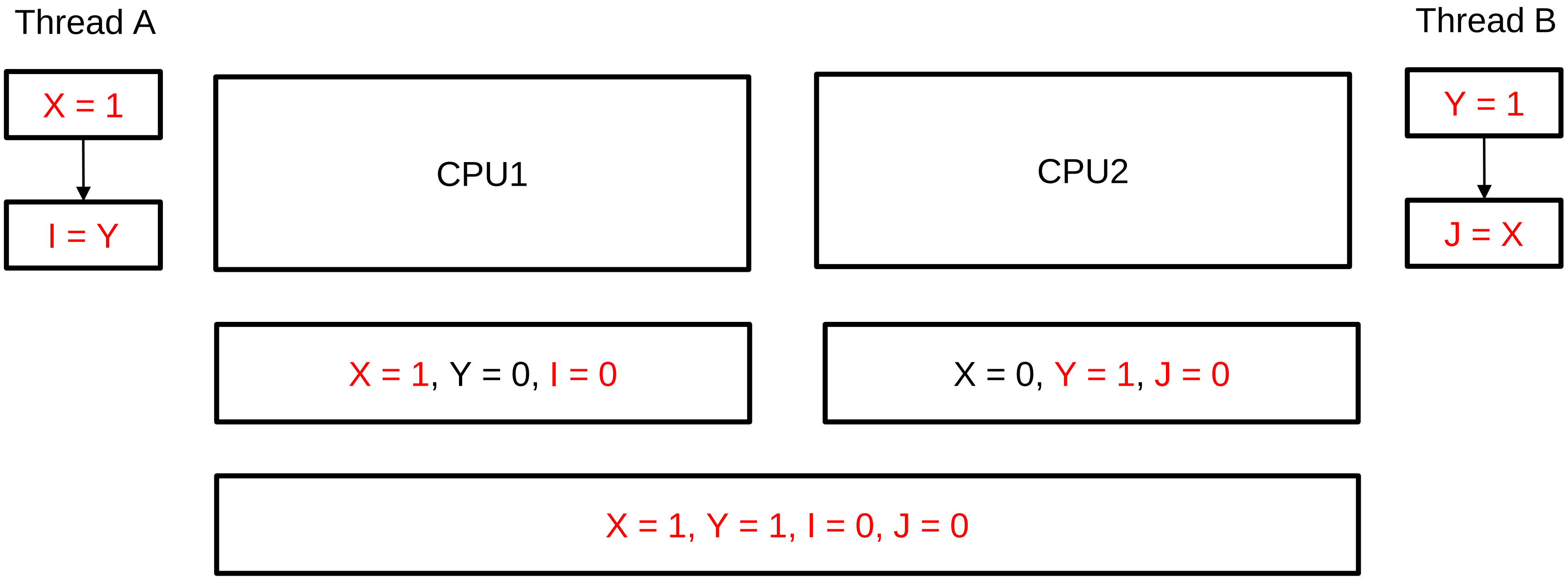
Quiz – Multicore case



Quiz – Multicore case

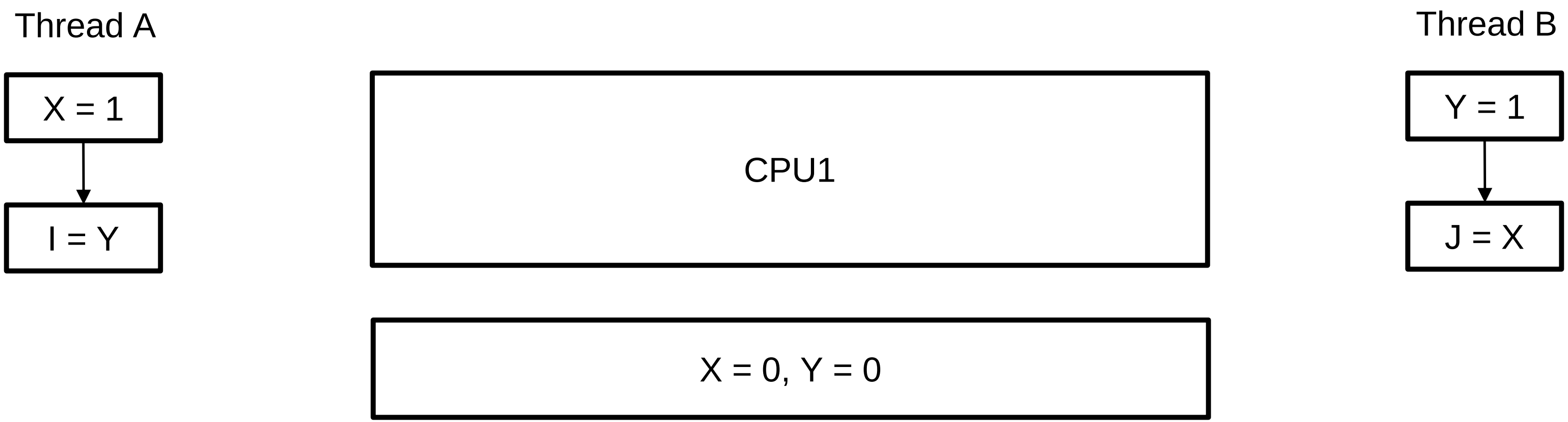


Quiz – Multicore case

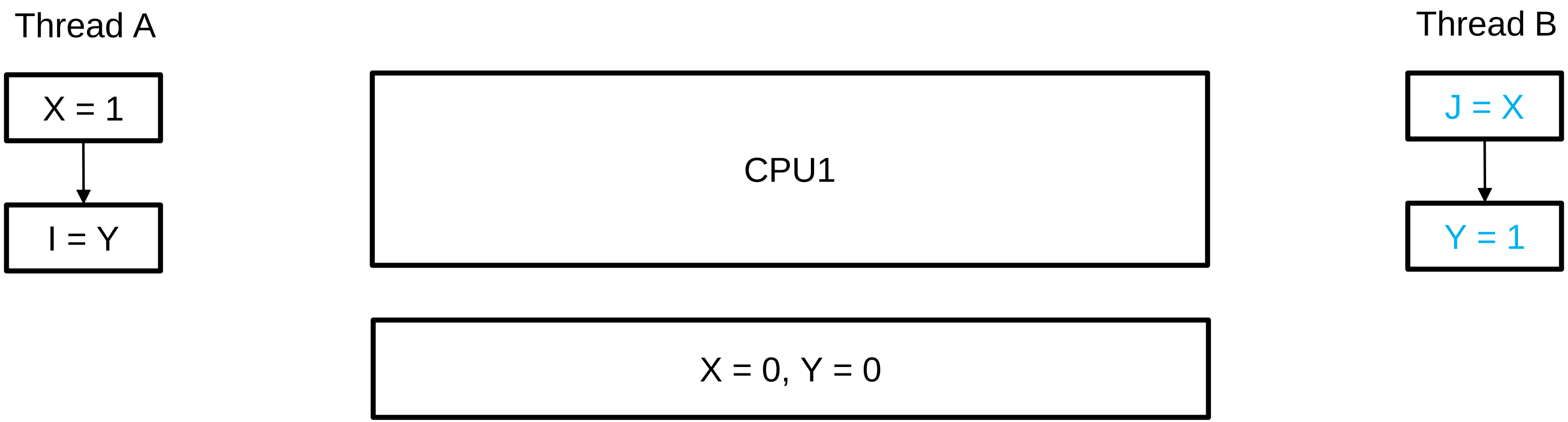


Eventually…

Quiz – Single core case

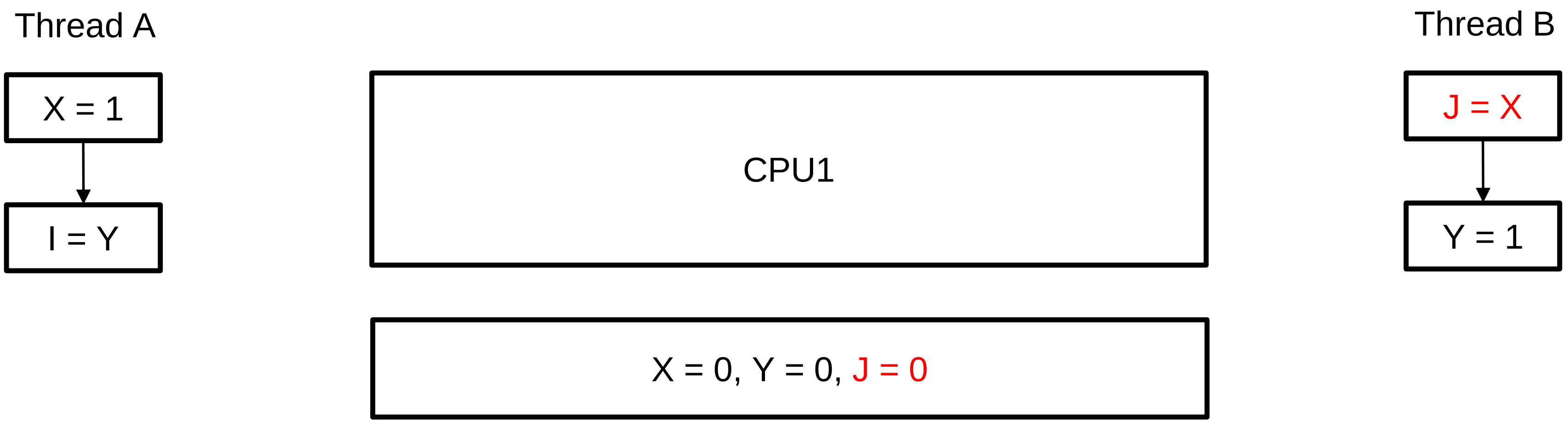


Quiz – Single core case

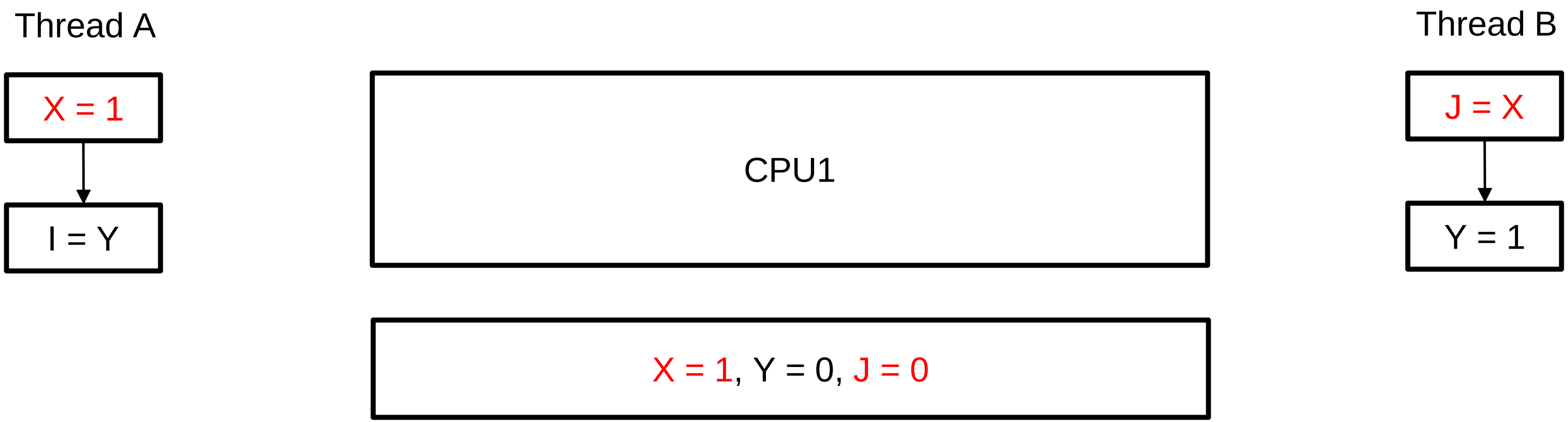


Compiler: It is more efficient to exchange these two unrelated instructions

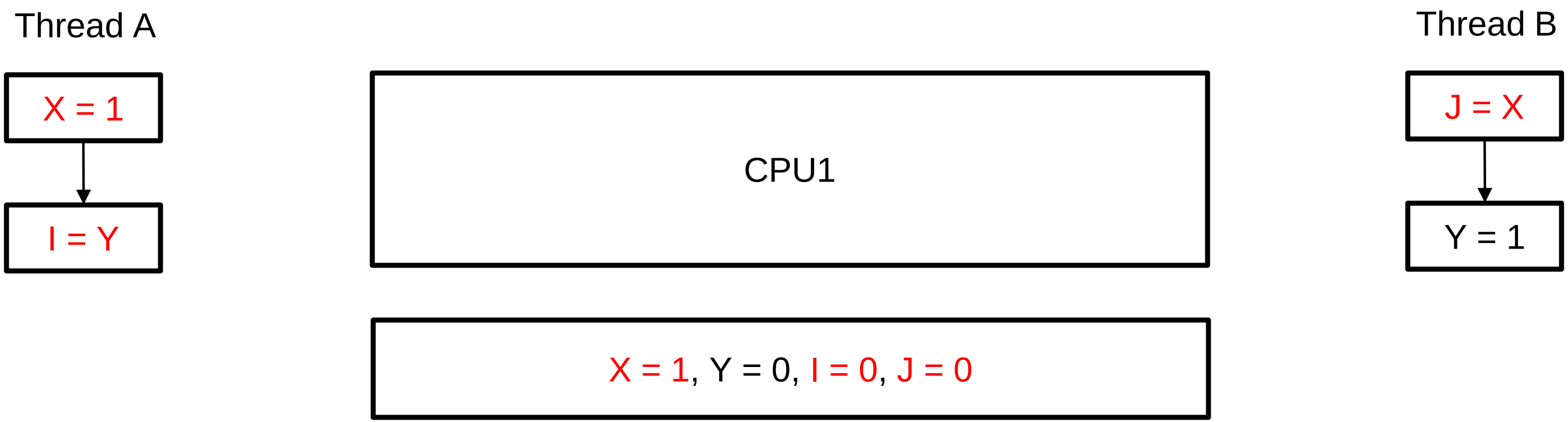
Quiz – Single core case



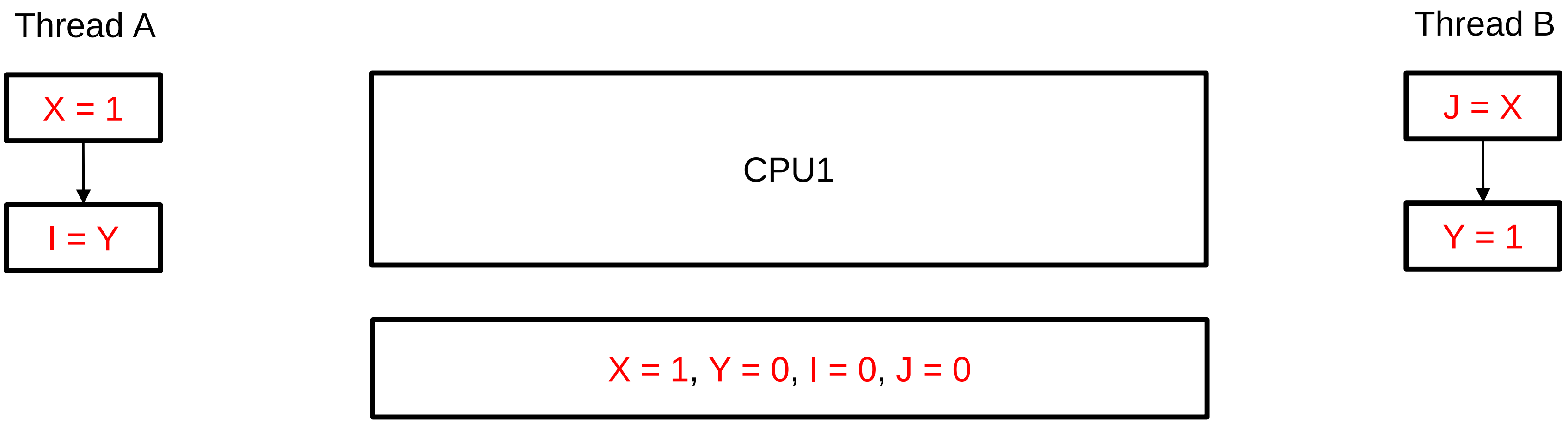
Quiz – Single core case



Quiz – Single core case



Quiz – Single core case



Java Memory Model

- Relaxed - Not even sequentially consistent!

Java Memory Model

- Relaxed - Not even sequentially consistent!
- Why? To accommodate compiler optimizations

Java Memory Model

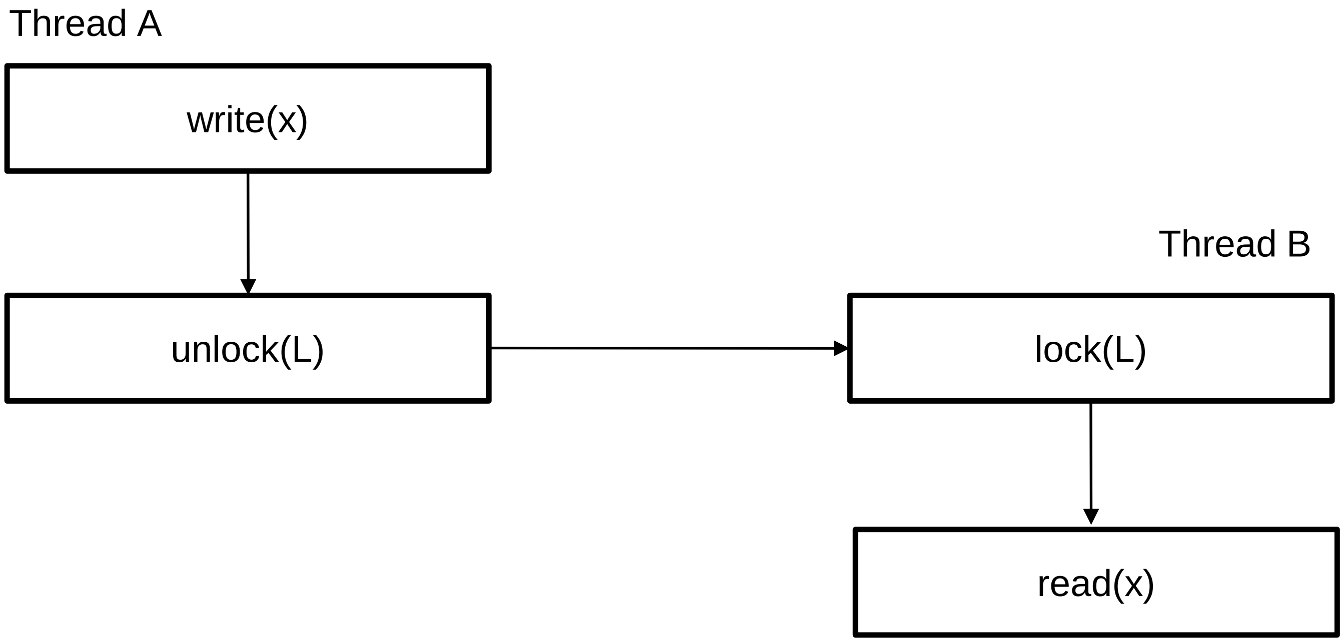
- Relaxed - Not even sequentially consistent!
- Why? To accommodate compiler optimizations

… all of which work by **caching** and/or **reordering memory reads–writes**

Java Memory Model

Executions can be made sequentially consistent on demand by using synchronization primitives and following a set of rules.

Synchronization



Volatile - Intuition

- **`volatile`** accesses do not count as data races
- the compiler does not touch **`volatile`** accesses
- forces reads and writes directly to memory

Volatile - Semantics are similar to locking

```
volatile int x;
void foo() {

  x = 1;

}
```

```
volatile int x;
void foo() {
  synchronized (x) {
    x = 1;
  }
}
```

Volatile - Semi Formal

- Accesses to volatile variables behave (almost) as if they are guarded by a “synchronized” block on itself, but

- variable can also be null
- cannot block
- works for primitive types

Volatile - Semi Formal

- Accesses to volatile variables behave (almost) as if they are guarded by a "synchronized" block on itself, but
  - variable can also be null
  - cannot block
  - works for primitive types
- each access goes directly to global memory

Volatile - Semi Formal

- Accesses to volatile variables behave (almost) as if they are guarded by a "synchronized" block on itself, but
  - variable can also be null
  - cannot block
  - works for primitive types
- each access goes directly to global memory
- volatile variables are linearizable

Volatile - Only individual accesses are “locked”

```
volatile int x;
void foo() {
  x++;
}
```

```
volatile int x; int tmp;
void foo() {
  synchronized (x) {
    tmp = x;
  }
  tmp = tmp + 1;
  synchronized (x) {
    x = tmp;
  }
}
```

Volatile - Typical Use Case

- **One writer** thread
- **Several reader** threads

Volatile - Typical Use Case

- **One writer** thread
- **Several reader** threads
- Commonly simple value updates:
  - set a flag
  - increment a counter, compute a max (single writer!)

Volatile - Typical Use Case

- **One writer** thread
- **Several reader** threads
- Commonly simple value updates:
  - set a flag
  - increment a counter, compute a max (single writer!)

- In case multiple writer threads: use atomics

Happens-before order

- Execution order within one thread established happens-before order

Happens-before order

- Execution order within one thread established happens-before order
- Lock release and subsequent lock acquire establish happens-before order

Happens-before order

- Execution order within one thread established happens-before order
- Lock release and subsequent lock acquire establish happens-before order
- Write to a volatile variable happens-before every subsequent read

Happens-before order

- Execution order within one thread established happens-before order
- Lock release and subsequent lock acquire establish happens-before order
- Write to a volatile variable happens-before every subsequent read
- Happens-before order is transitive

More formal treatment

The Java Memory Model*

Jeremy Manson and William Pugh
Department of Computer Science
University of Maryland, College Park
College Park, MD
{jmanson, pugh}@cs.umd.edu

Sarita V. Adve
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana-Champaign, IL
sadve@cs.uiuc.edu

ABSTRACT

This paper describes the new Java memory model, which has been revised as part of Java 5.0. The model specifies the legal behaviors for a multithreaded program; it defines the semantics of multithreaded Java programs and partially determines legal implementations of Java virtual machines and compilers.

The new Java model provides a simple interface for correctly synchronized programs – it guarantees sequential consistency to data-race-free programs. Its novel contribution is requiring that the behavior of incorrectly synchronized programs be bounded by a well defined notion of causality. The causality requirement is strong enough to respect the

Meanings of Programs]: Operational Semantics

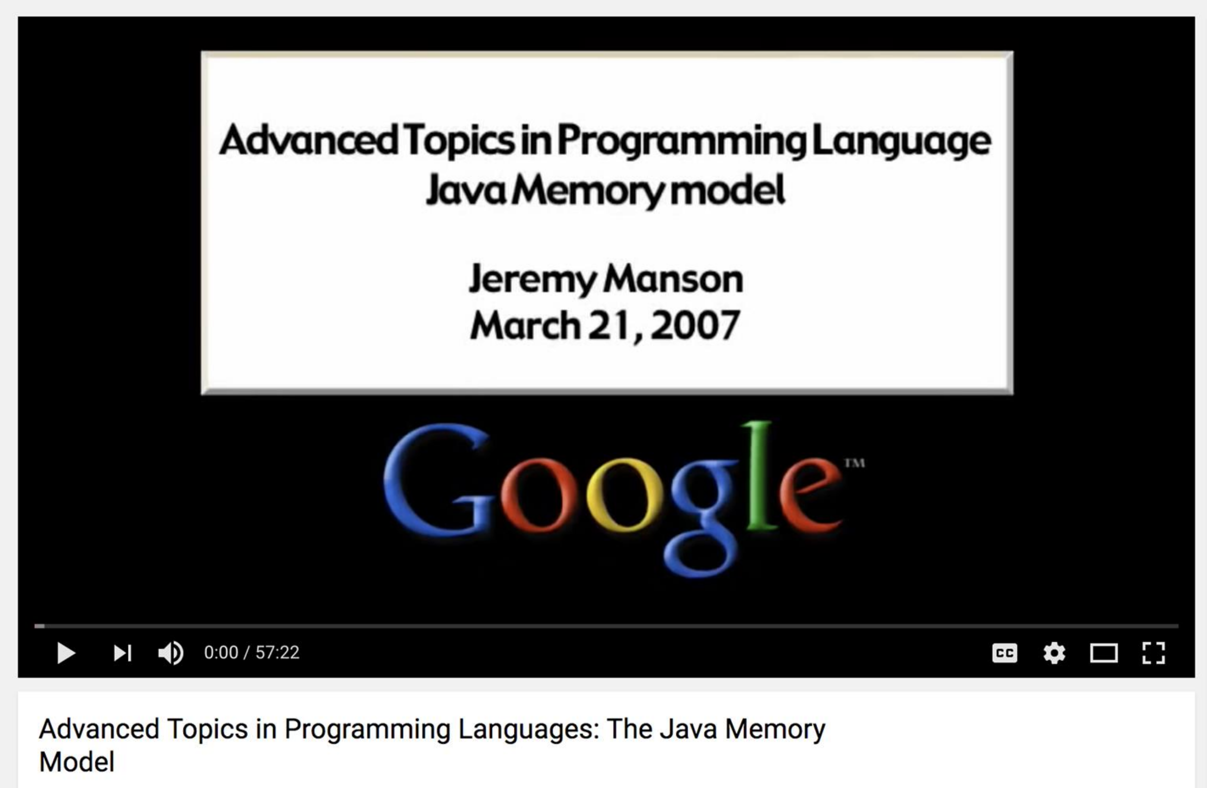
General Terms: Design, Languages

Keywords: Concurrency, Java, Memory Model, Multithreading

1. INTRODUCTION

The memory model for a multithreaded system specifies how memory actions (e.g., reads and writes) in a program will appear to execute to the programmer, and specifically, which value each read of a memory location may return. Every hardware and software interface of a system that admits multithreaded access to shared memory requires a memory

More informal treatment



https://www.youtube.com/watch?v=WTVooKLLVT8

Java Language Specification

.

ORACLE

Java SE > Java SE Specifications > Java Language Specification

Multi-valent states

Consensus. Multivalent states

- **Precondition:** every participant proposes a value (not known to others)
- **Postcondition:** all participants decide on the same value (known to others)

- ***Conclusion: there must be a transition between one and the other.***

Consensus. Multivalent states

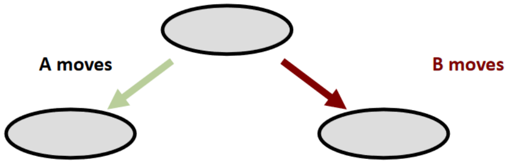
- **Precondition:** every participant proposes a value (not known to others) – ***multivalent***
- **Postcondition:** all participants decide on the same value (known to others) **- *univalent***

- ***Conclusion: there must be a transition between one and the other – critical state***

Consensus. Multivalent states

- **Precondition:** every participant proposes a value (not known to others) – ***multivalent***
- **Postcondition:** all participants decide on the same value (known to others) **- *univalent***

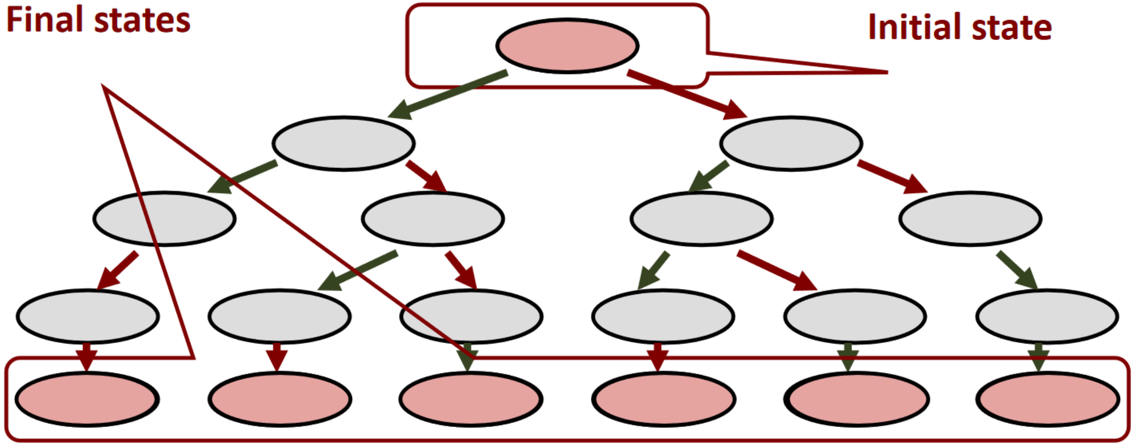
- ***Conclusion: there must be a transition between one and the other – critical state***



Consensus. Multivalent states

- **Precondition:** every participant proposes a value (not known to others) – ***multivalent***
- **Postcondition:** all participants decide on the same value (known to others) **-** ***univalent***

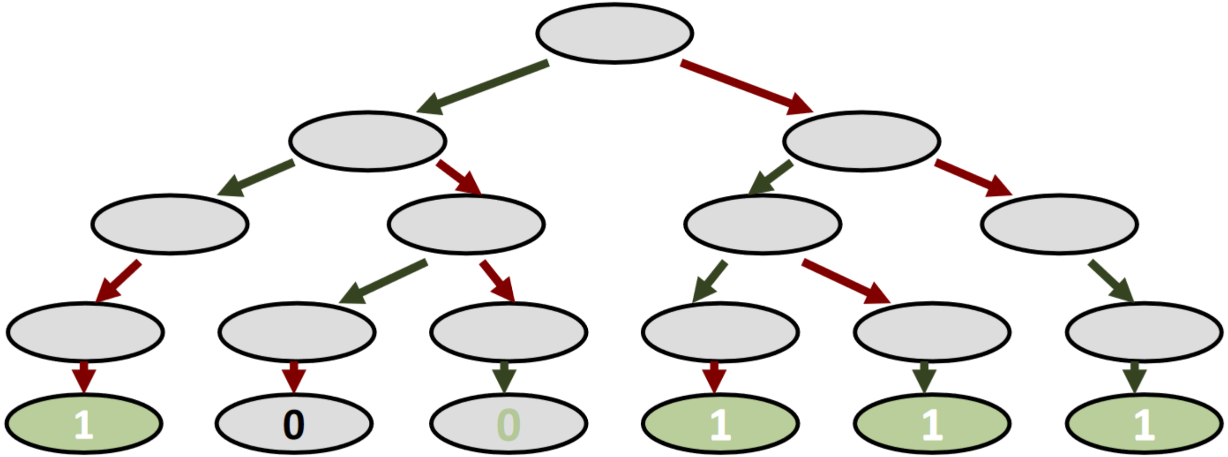
- ***Conclusion: there must be a transition between one and the other – critical state***



Consensus. Multivalent states

- **Precondition:** every participant proposes a value (not known to others) – ***multivalent***
- **Postcondition:** all participants decide on the same value (known to others) **-** ***univalent***

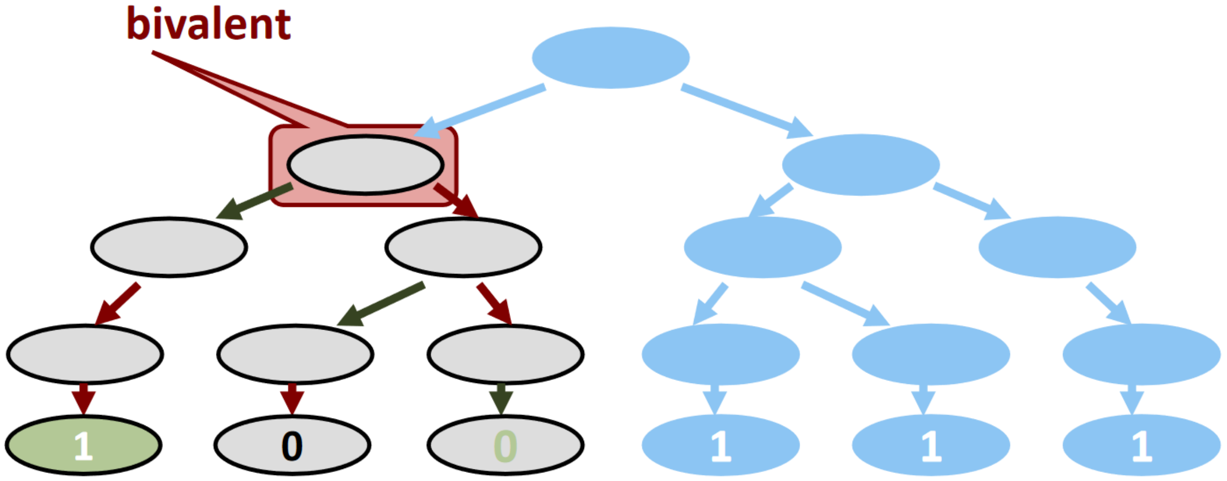
- ***Conclusion: there must be a transition between one and the other – critical state***



Consensus. Multivalent states

- **Precondition:** every participant proposes a value (not known to others) – ***multivalent***
- **Postcondition:** all participants decide on the same value (known to others) **- *univalent***

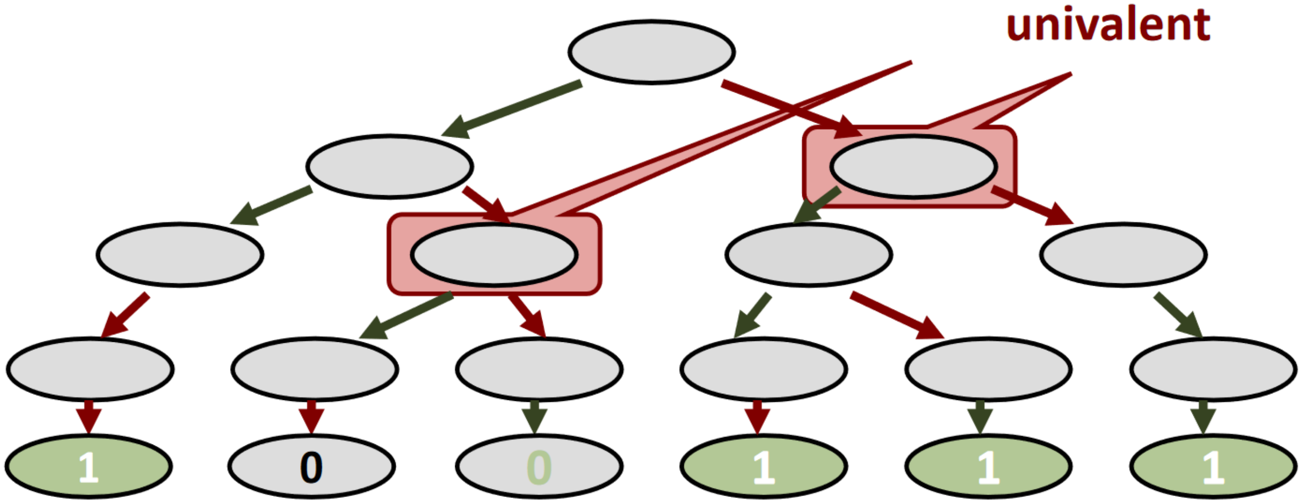
- ***Conclusion: there must be a transition between one and the other – critical state***



Consensus. Multivalent states

- **Precondition:** every participant proposes a value (not known to others) – ***multivalent***
- **Postcondition:** all participants decide on the same value (known to others) **- *univalent***

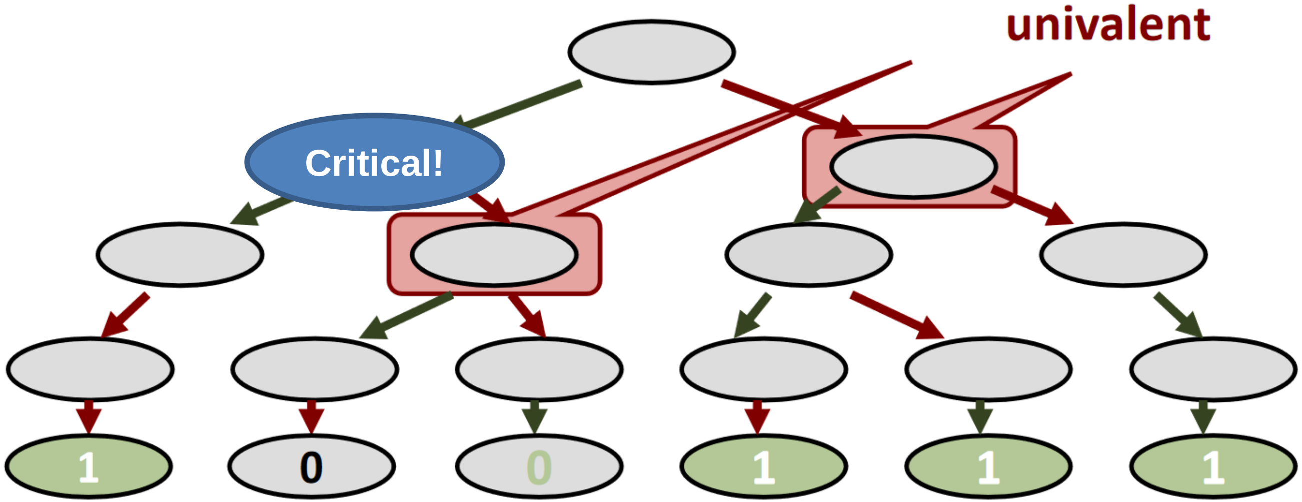
- ***Conclusion: there must be a transition between one and the other – critical state***



Consensus. Multivalent states

- **Precondition:** every participant proposes a value (not known to others) – ***multivalent***
- **Postcondition:** all participants decide on the same value (known to others) **-** ***univalent***

- ***Conclusion: there must be a transition between one and the other – critical state***



Assignment 12

Exercises

Exercise 1 – Wait-free implies lock free

Explain why a valid wait-free consensus protocol cannot use locks.

Exercises

Execise 2 – Valence states

Assume N=2 and inputs are either 0 or 1 for each agent. Thus in the initial state of any consensus we are in a bivalent state (bivalent = the output can be 0 or 1). However, at termination all agents have agreed on a single value, thus we are in a univalent state. Explain why there is a finite number of bivalent states in any wait-free consensus protocol.

Exercises

Execise 2 – Valence states

Assume N=2 and inputs are either 0 or 1 for each agent. Thus in the initial state of any consensus we are in a bivalent state (bivalent = the output can be 0 or 1). However, at termination all agents have agreed on a single value, thus we are in a univalent state. Explain why there is a finite number of bivalent states in any wait-free consensus protocol.

Exercises

Execise 2 – Valence states

Assume N=2 and inputs are either 0 or 1 for each agent. Thus in the initial state of any consensus we are in a bivalent state (bivalent = the output can be 0 or 1). However, at termination all agents have agreed on a single value, thus we are in a univalent state. Explain why there is a finite number of bivalent states in any wait-free consensus protocol.

Requirements on consensus protocol

- wait-free: consensus returns in finite time for each thread
- consistent: all threads decide the same value
- valid: the common decision value is some thread's input

Exercises

Exercise 3 – Consensus among prisoners

Imagine there are 100 people in a prison. Each day the warden picks a prisoner (each prisoner with the probability 1/100). The prisoner is led to a room with a light that he can turn on or off. Initially the light is turned off. After the prisoner was in the room he can state ”by now every prisoner was in the room at least once”. If this statement is made and it is true, all prisoners are released. If the statement is made and it is false, all prisoners are shot. Devise a strategy that the prisoners can follow to make sure they get released some day in the future with absolute certainty (no other communication is allowed).

Exercises

Exercise 4 – Implementing two thread consensus

Assume you have a machine with atomic registers and an atomic test-and-set operation with the following semantics (X is initialized to 1):

```
int TAS() {
  res = X;
  if (res = 1) {
    X = 0
  }
  return(res)
}
```

Implement a two-process consensus protocol using TAS() and atomic registers.

Exercises

Exercise 4 – Implementing two thread consensus

Assume you have a machine with atomic registers and an atomic test-and-set operation with the following semantics (X is initialized to 1):

```
int TAS() {
  res = X;
  if (res = 1) {
    X = 0
  }
  return(res)
}
```

Implement a two-process consensus protocol using TAS() and atomic registers.

Generic consensus protocol

```
public abstract class ConsensusProtocol<T> implements Consensus<T> {
  protected T[] proposed = (T[]) new Object[N];
  // announce my input value to the other threads
  void propose(T value) {
    proposed[ThreadID.get()] = value;
  }
  // figure out which thread was first
  abstract public T decide(T value);
}
```

Figure 5.6 The generic consensus protocol.

Generic consensus protocol

```
public abstract class ConsensusProtocol<T> implements Consensus<T> {
  protected T[] proposed = (T[]) new Object[N];
  // announce my input value to the other threads
  void propose(T value) {
    proposed[ThreadID.get()] = value;
  }
  // figure out which thread was first
  abstract public T decide(T value);
}
```

Figure 5.6 The generic consensus protocol.

Implement decide() method and constructor
(See consensus using FIFO)

Exercises

Exercise 5 – Linearizability

Which of the following scenarios are lineraly consistent, assuming `s` is a stack? Either mark the point of linearization or explain why it is not linearly consistent.