MULTICORE PROGRAMMING

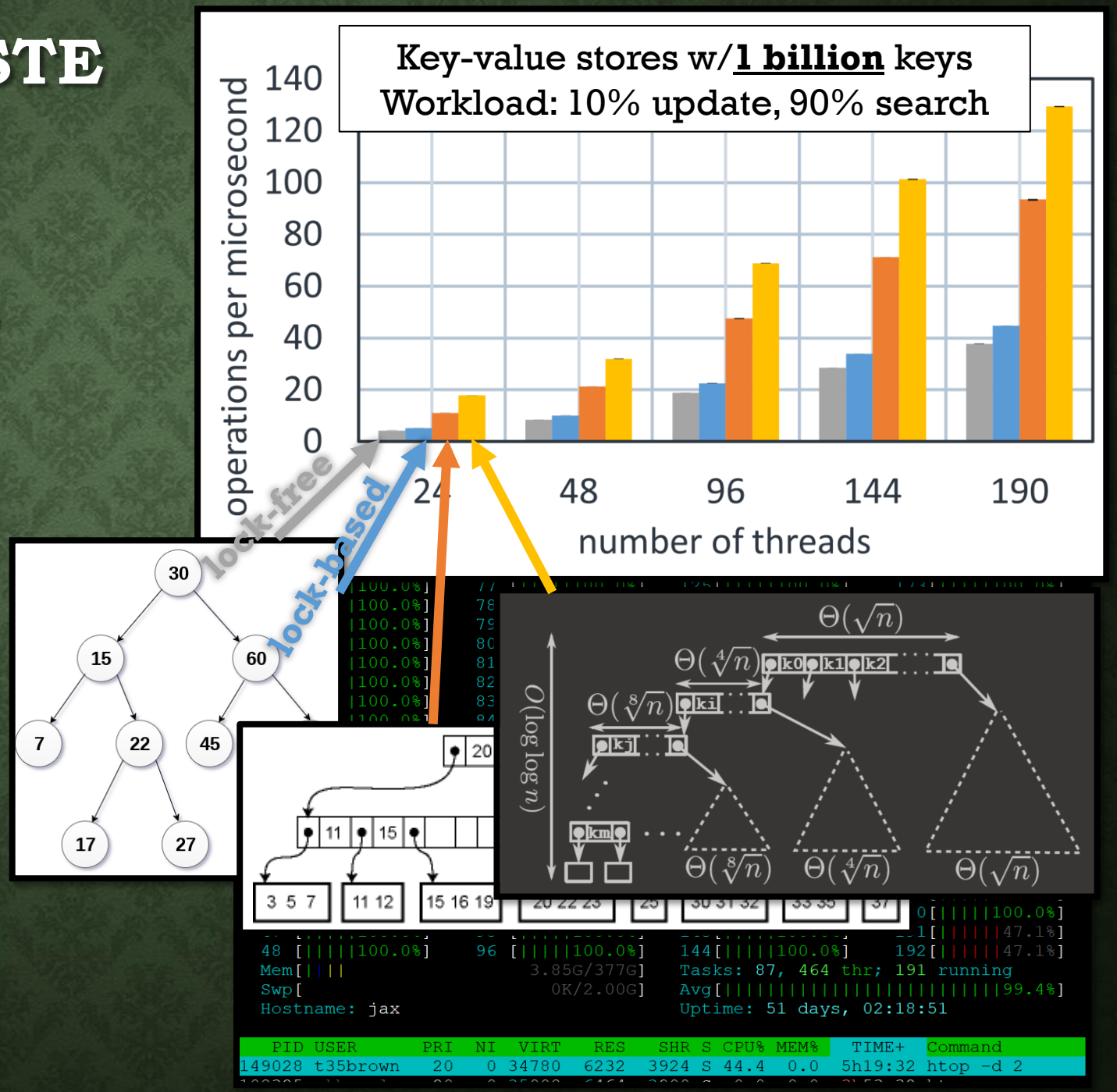
Linearizability: defining correctness for concurrency

Lecture 1

Trevor Brown

MY RESEARCH: A TASTE

- **Big multicore systems**
  - 192 thread Intel
  - 144 thread Intel, 9x 32-thread cluster, soon 256 thread AMD…
  - **How to harness them effectively?**
- Fast concurrent data structures
  - Lock-free balanced search trees
- Synchronization primitives
- Memory allocation & reclamation
- Transactional memory
- Non-volatile memory



THIS COURSE

- Emphasis on both **t**heory and practice
  - **Leaning towards practice**
  - Correctness is hard
  - Theory is needed to make things rigorous
  - **But,** concurrency is about performance
  - So, practice **must** inform theory!
- Less focus on traditionally taught concurrent programming material
  - Attempting to teach the unspoken/unwritten intuition of experts
  - Hoping to reach the bleeding edge
- Covering C++ (Java is similar and often easier – but less control!)

BACKGROUND

Introductory material

CONCURRENCY AND THREADS

- Threads
  - Building block for concurrency
  - Relatively lightweight
    (overhead in ~1000s-10000s of cycles)
  - Threads can share the same memory space
    - Each thread has private memory
    - And access to shared memory (via reads & writes)
      - Read and write are **atomic**
      - Cannot see partially written values
  - Use to call a function **asynchronously**
  - Can **synchronize** with **join**

Knowing what these are…

Doesn't mean you can do this :)

FORK/JOIN: COMMON DESIGN PATTERN

- Identify work that can be **divided** between threads
- **Fork** many threads, each doing some share of the work
- **Join** the threads (synchronize)
- Repeat if necessary

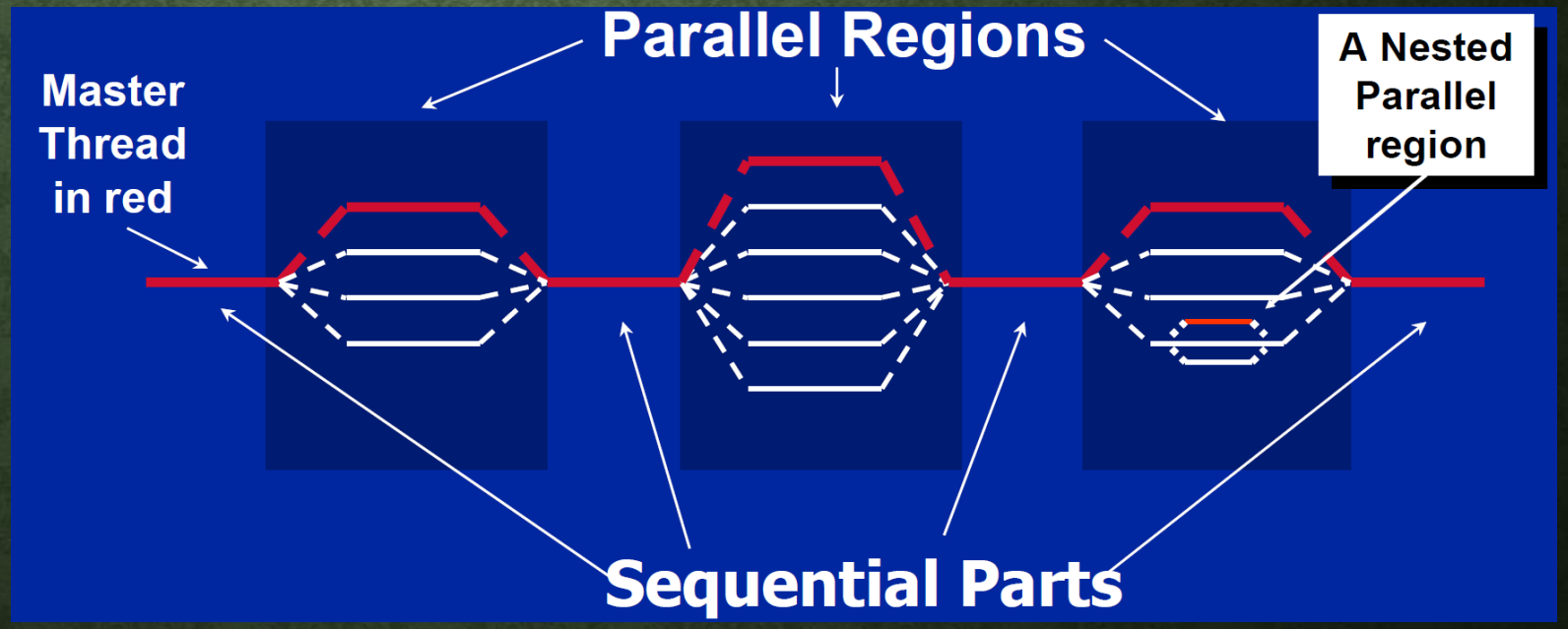


Diagram shamelessly stolen; see slide notes

BASIC THREADING EXAMPLES: #0

```cpp
#include <iostream>
#include <thread>
void func1() {
    std::cout<<"one\n";
}
void func2() {
    std::cout<<"two\n";
}
int main(int argc, char** argv) {
    std::thread t1 (func1);
    std::thread t2 (func2);
    t1.join();
    t2.join();
    return 0;
}
```

```
one
two

one
two

one
two

one
two

one
two

one
two
```

```
one
two

one
two

one
two

one
two

one
two

one
two
```

Five inversions
after 1000 runs

■ ■ ■

Thread 1 starts first...
biasing the line order

BASIC THREADING EXAMPLES: #1

```cpp
#include <iostream>
#include <thread>
void func1() {
    for (int i=0;i<10;++i)
        std::cout<<"one"<<std::endl;
}
void func2() {
    for (int i=0;i<10;++i)
        std::cout<<"two"<<std::endl;
}
int main(int argc, char** argv) {
    std::thread t1 (func1);
    std::thread t2 (func2);
    t1.join();
    t2.join();
    return 0;
}
```

```
onetwo
one
one
one
one
one
two
two
two
two
two
two
two
two

one
one
one
one
two
```

```
one
twoone
one
one
one
one
one
one
one
one

two
two
two
two
two
two
two
two
two
two
```

```
one
one
one
one
one
one
one
one
one
twoone

two
two
two
two
two
two
two
two
two
two
```

```
one
one
one
onetwo

two
two
two
two
two
two
two
one
one
one
one
one
```

BASIC THREADING EXAMPLES: #2

```cpp
#include <iostream>
#include <thread>
void func1() {
    for (int i=0;i<1000;++i)
        std::cout<<".";
}
void func2() {
    for (int i=0;i<1000;++i)
        std::cout<<"X";
}
int main(int argc, char** argv) {
    std::thread t1 (func1);
    std::thread t2 (func2);
    t1.join();
    t2.join();
    std::cout<<std::endl;
    return 0;
}
```

EX3: USING THIS TO VISUALIZE THREAD SCHEDULING

```cpp
#include <iostream>
#include <thread>
void func(int intensity) {
    for (int i=0;i<1000;++i)
        std::cout<<".:-=+*#&%@"[intensity];
}
int main(int argc, char** argv) {
    const int n = 10;
    std::thread * t[n];
    for (int i=0;i<n;++i)
        t[i] = new std::thread(func, i);
    for (int i=0;i<n;++i)
        t[i]->join();
    return 0;
}
```

```
1  #include <iostream>
2  #include <thread>
3  bool x = 0;
4  void func1() {
5      while (!x) { /* busy-wait */ }
6      std::cout<<"1";
7  }
8  void func2() {
9      std::cout<<"2";
10     x = true;
11 }
12 int main(int argc, char** argv) {
13     std::thread t1 (func1);
14     std::thread t2 (func2);
15     t1.join();
16     t2.join();
```

BASIC EXAMPLES…

- … can lead to hard questions!
- What happens in this code?
  - At –O0 (no optimization), print "21"
  - **At –O3 it runs forever!**
- What's happening here?
- Compiler optimizations often **break** multithreaded code!

-O0 VS -O3

```cpp
#include <iostream>
#include <thread>
bool x = 0;
void func1() {
    while (!x) { /* busy-wait */ }
    std::cout<<"1";
}
void func2() {
    std::cout<<"2";
    x = true;
}
int main(int argc, char** argv) {
    std::thread t1 (func1);
    std::thread t2 (func2);
    t1.join();
    t2.join();
```

Repeatedly read and test x **inside** the loop

Read and test x **once before an empty loop**

ASM generated using GCC 9.1 via https://godbolt.org/

-O0

```
56 func1():
57         push    rbp
58         mov     rbp, rsp
59 .L11:
60         movzx   eax, BYTE PTR x[rip]
61         test    al, al
62         jne     .L10
63         jmp     .L11
64 .L10:
65         mov     esi, OFFSET FLAT:.LC0
66         mov     edi, OFFSET FLAT:_ZSt4cout
67         call    std::basic_ostream<char, std:
```

-O3

```
28 func1():
29         cmp     BYTE PTR x[rip], 0
30         jne     .L9
31 .L10:
32         jmp     .L10
33 .L9:
34         mov     edx, 1
35         mov     esi, OFFSET FLAT:.LC1
36         mov     edi, OFFSET FLAT:_ZSt4cout
37         jmp     std::basic_ostream<char, std:
```

Important principle!

THE ISSUE

- Compiler doesn't know **x may be changed by other threads**
- In a single-threaded execution it wouldn't
- Solution
  - Use **C++ Atomics** for **x**
  - Informs the compiler that **x** may change
  - (Before C++ atomics, we used **volatile** for this)
- **Principle:** any variable written by at least one thread, and read by at least one **other** thread, should be an **atomic<T>** type

USING ATOMIC<BOOL>

```cpp
#include <iostream>
#include <thread>
#include <atomic>
std::atomic<bool> x (0);
void func1() {
    while (!x) { /* busy-wait */ }
    std::cout<<"1";
}
void func2() {
    std::cout<<"2";
    x = true;
}
int main(int argc, char** argv) {
    std::thread t1 (func1);
    std::thread t2 (func2);
    t1.join();
    t2.join();
```

Repeatedly read and test x **inside** the loop

-O0

```
func1():
        push    rbp
        mov     rbp, rsp
.L18:
        mov     edi, OFFSET FLAT:x
        call    std::atomic<bool>::operator bool()
        xor     eax, 1
        test    al, al
        je      .L17
        jmp     .L18
.L17:
        mov     esi, OFFSET FLAT:.LC0
        mov     edi, OFFSET FLAT:_ZSt4cout
        call    std::basic_ostream<char, std::char_
```

Repeatedly read and test x **inside** the loop… **correct!**

-O3

```
func1():
.L9:
        movzx   eax, BYTE PTR x[rip]
        test    al, al
        je      .L9
        mov     edx, 1
        mov     esi, OFFSET FLAT:.LC1
        mov     edi, OFFSET FLAT:_ZSt4cout
        jmp     std::basic_ostream<char, std
```

Important definition!

DEFINITION: (SHARED) OBJECT

- An **object** has some **abstract state** and offers one or more **operations**
- An **operation** has an **invocation** (possibly **taking some arguments**) and a **response** (possibly **returning a value**)
- An object can be accessed simultaneously by many threads

EXAMPLE: COUNTER OBJECT

Abstract state! We are not saying anything about the **implementation**.

- Stores an integer value, initially zero
- Operations…
  - Increment
    - Increase the value by one and return the old value
  - Read
    - Return the current value

How to define semantics? Want to **avoid implementation details…**

Again, we are talking about the **abstract state**

Not any particular representation in memory…

MOTIVATION

- Why might you want a counter?
  - To keep track of the **size** of a **concurrent hash table**
    - Counter is incremented whenever an element is inserted
    - Counter is read to decide whether the table should be **expanded**
  - Suppose accesses to the hash table are spread out and highly concurrent
    - Then the hash table is **not** a concurrency bottleneck
  - Counter should not be a concurrency bottleneck either!
- Why are **we** studying counters?
  - As a **vehicle** to understand concurrency, correctness, performance and systems

NAÏVE IMPLEMENTATION OF A COUNTER

- Using **reads** and **writes**
- Increment(c)
  - x = Read(c)
  - Write(c, x + 1)
  - Return x
- Get(c)
  - x = Read(c)
  - Return x

```
17  class counter1 {
18  private:
19      int v;
20  public:
21      counter1() { v = 0; }
22      int increment(int threadID) { return v++; }
23      int get() { return v; }
24  };
```

This is really a **read** followed by a **write**…

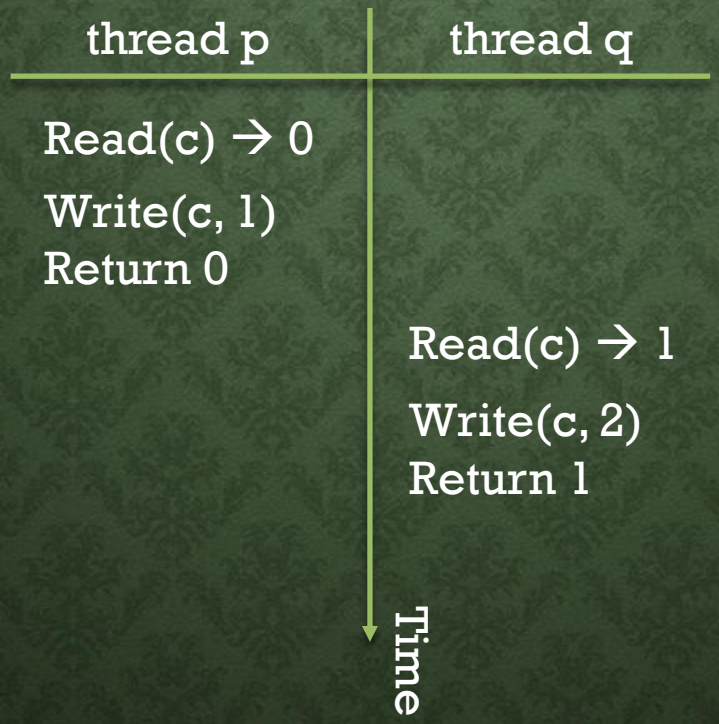
Important definition!

EXECUTION OF AN ALGORITHM

- Execution **E** is a sequence of alternating **configurations** and **steps**
  - $C_1\, s_1\, C_2\, s_2\, C_3\, s_3 \ldots$
- Configurations and steps capture how the system changes over time
- A **configuration** describes the state of the system
  - Contents of memory
  - Program counters of all threads
- A **step** is a **read** or **write** to shared memory
  - (Or an atomic synchronization primitive --- more on that later)

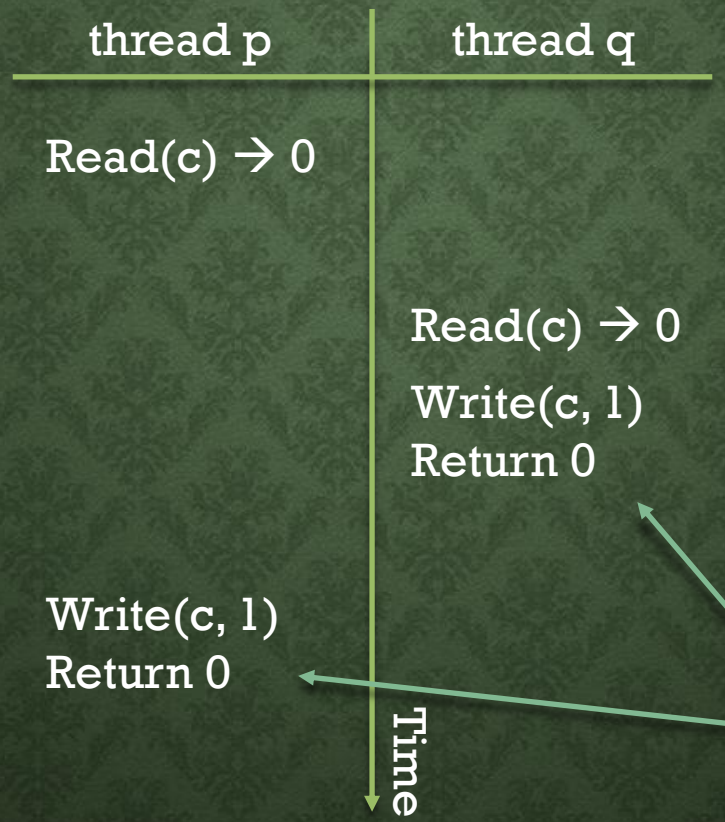
EXAMPLE EXECUTION 1: OUR NAÏVE COUNTER IMPLEMENTATION

- **Suppose two threads p and q both access the same counter c**



After 2 increments
Final counter value = 2

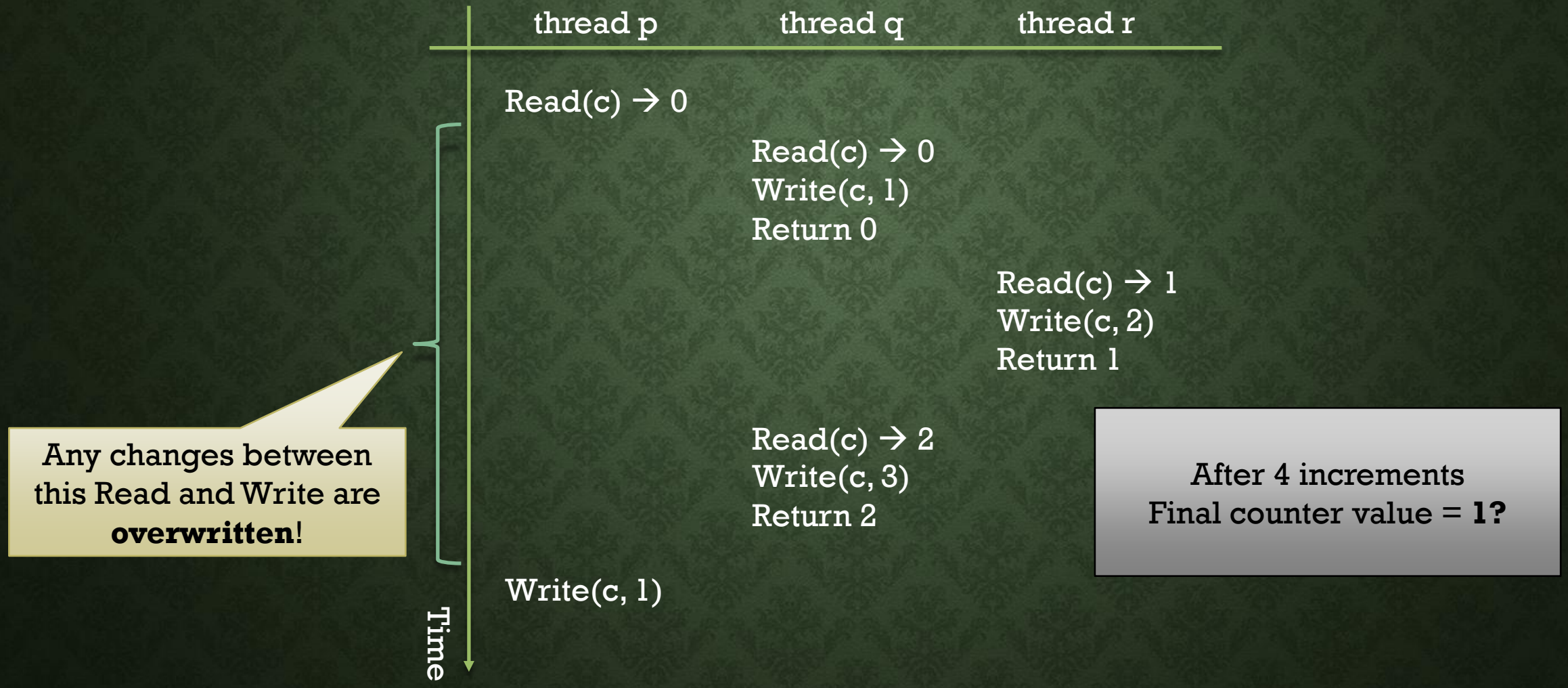
EXAMPLE EXECUTION 2



After 2 increments
Final counter value = **1?**
Incorrect!

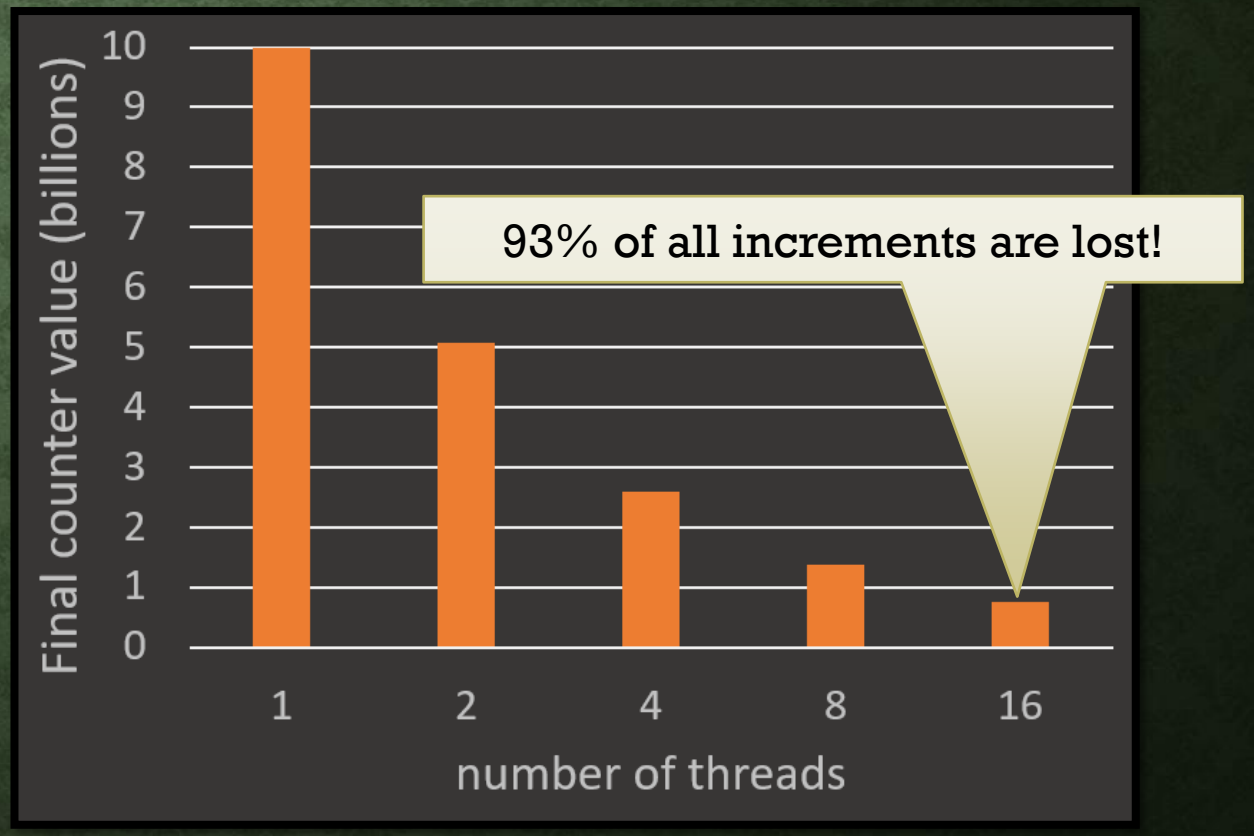
Same return values?
Incorrect!

EXAMPLE EXECUTION 3



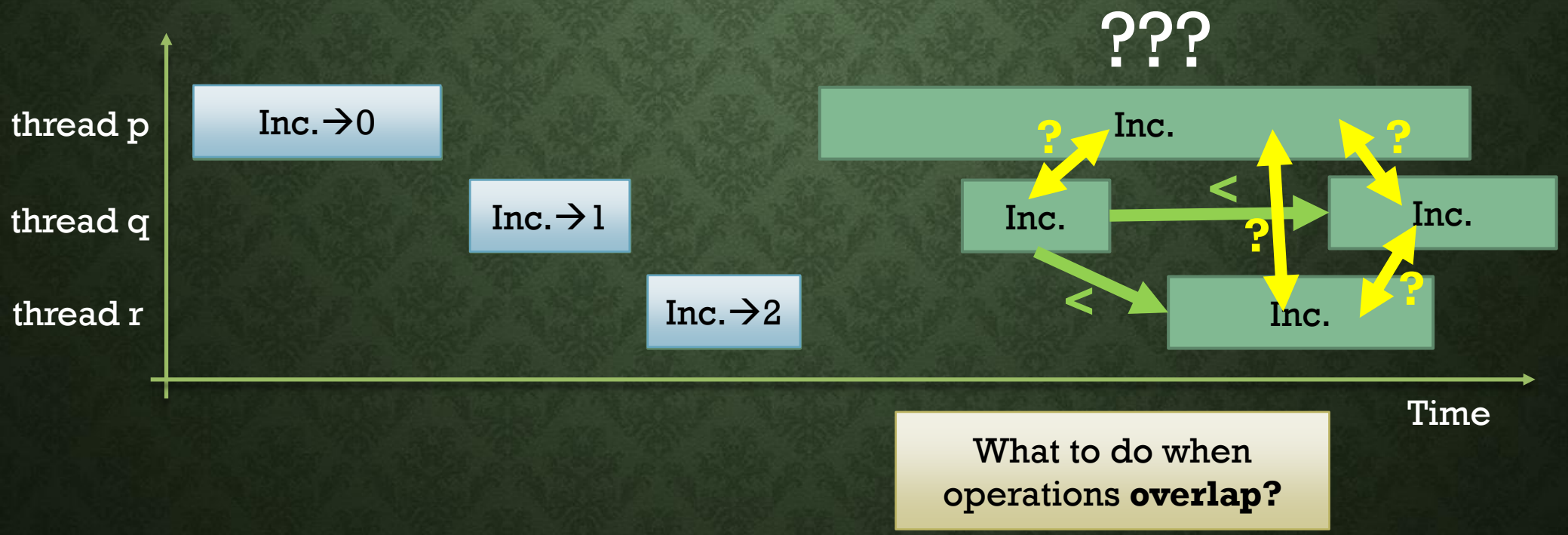
HOW BAD IS THIS NAÏVE COUNTER'S ACCURACY IN PRACTICE?

- Simple experiment
  - n threads share one counter
  - Each thread performs (10 billion / n) increments
- What should the **final counter value** be at the end of the execution?



WHAT DOES IT MEAN TO BE CORRECT?

- In the **counter accuracy** experiment
  - Obvious: at the end, counter should = 10 billion
  - Not obvious: during the experiment, **what should each Increment() return**?



WHAT DOES IT MEAN TO BE CORRECT?

- Correctness condition: **Linearizability**
- High level idea
  - **Every** concurrent execution **E** has an **equivalent** **sequential execution**
    - I.e., **E** represents a **sequential execution that could happen**!
    - Implies: **E cannot** do anything that is **impossible** in a sequential execution
  - Lifts an object's sequential definition into the concurrent setting
    - Concurrent behaviour is *defined by valid sequential behaviour*

CRITICAL definition!

DEFINING LINEARIZABILITY

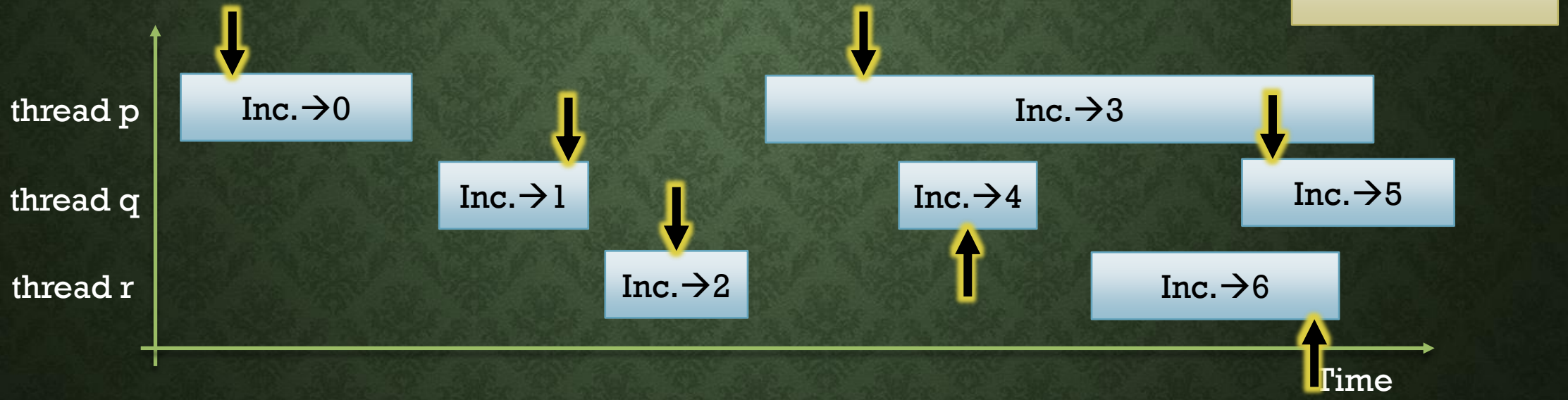
- A **concurrent execution** is **linearizable** if
  - each operation has a **linearization point** *during* the operation, such that
  - if all operations appear to occur **instantly** at their linearization points, then they behave exactly as defined in the **sequential** abstract data type
- An **object** is linearizable if **all possible executions** are linearizable

Execution 1

IS THIS LINEARIZABLE?

- I.e., can we choose **linearization points** during each operation such that all operations respect the sequential ADT?

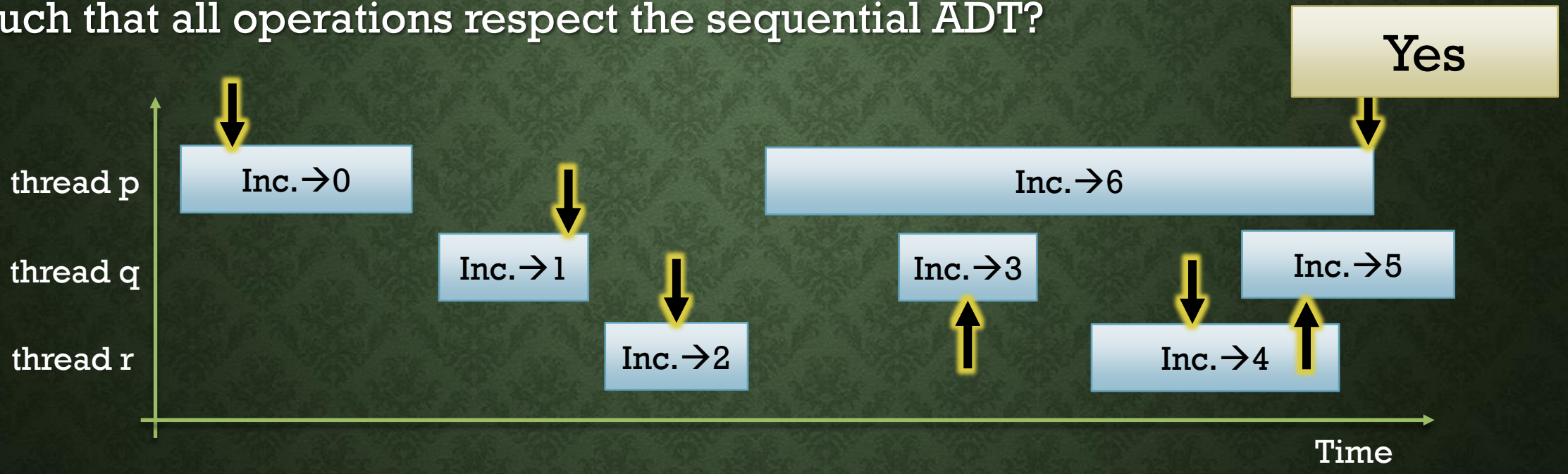
Yes



Execution 2

IS THIS LINEARIZABLE?

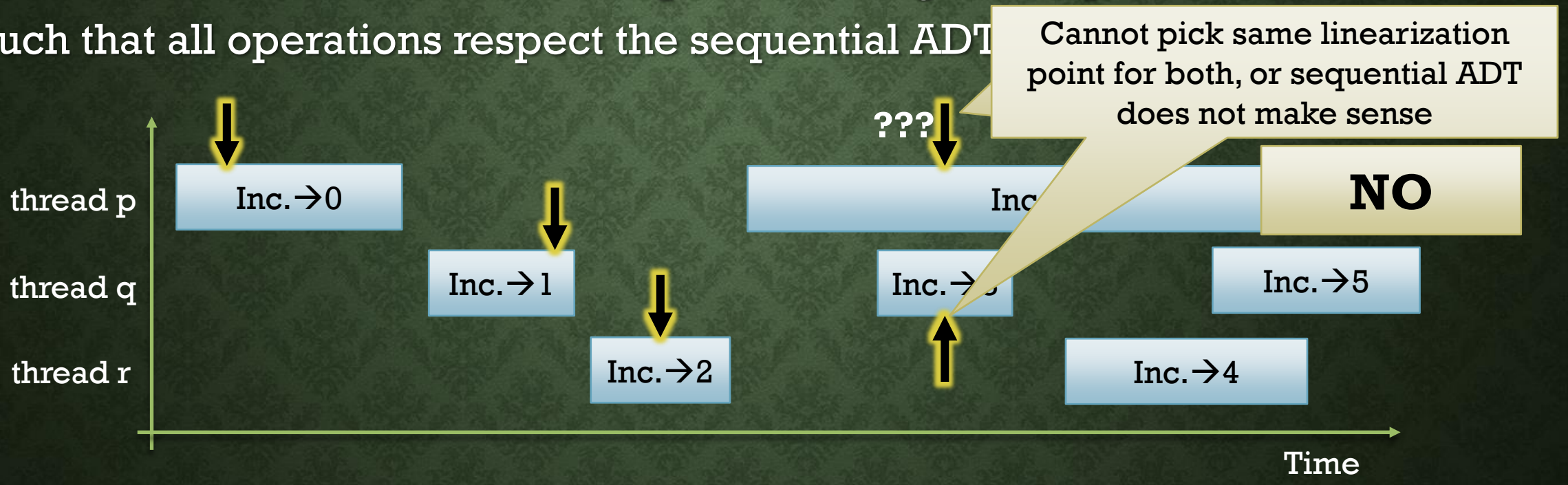
- I.e., can we choose **linearization points** during each operation such that all operations respect the sequential ADT?



Execution 3

IS THIS LINEARIZABLE?

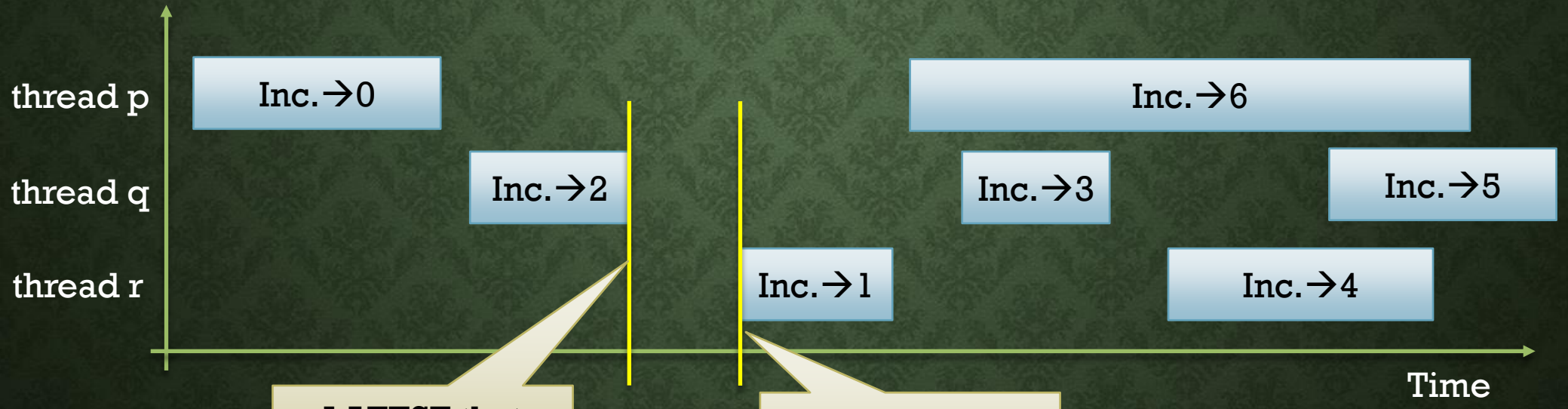
- I.e., can we choose **linearization points** during each operation such that all operations respect the sequential ADT



Execution 4

IS THIS LINEARIZABLE?

- I.e., can we choose **linearization points** during each operation such that all operations respect the sequential ADT?



LATEST that "Inc.→2" can linearize

EARLIEST that "Inc.→1" can linearize

So "Inc.→1" must be linearized after "Inc.→2"
(violating the sequential ADT)

NO

ASSIGNMENT 1

- Released now
- Due in **one week**
- Goals:
  - Get your development environment set up
  - Connect to remote servers that you can use for development (for this class)
  - **Learn Linux tools**
  - **Practice turning human-readable text output into experimental graphs**
  - Learn how to submit your programs to Marmoset
  - Learn how to submit written parts to Crowdmark

I *hope* you can login… but if you can't, we'll figure it out.

COURSE RESOURCES

- No official textbook (just slides and some supplementary papers)
  - That said, The Art of Multiprocessor Programming by Herlihy and Shavit may be helpful
- See Piazza for slides, assignments, announcements, questions
- See Crowdmark for submitting ***written parts*** of assignments
- See Marmoset for submitting ***code***

SUMMARY

- Important definitions
  - Threading model (shared memory)
  - (Shared) object
  - Execution (of an algorithm)
  - Linearizability (of executions and objects)
- Important system concepts
  - Threading via std::thread
  - C++ atomics (principle: when to use atomics)
- Assignment 1 due in a week