CS 314 Principles of Programming Languages

Lecture 22: Type Systems, Concurrent Data Structure

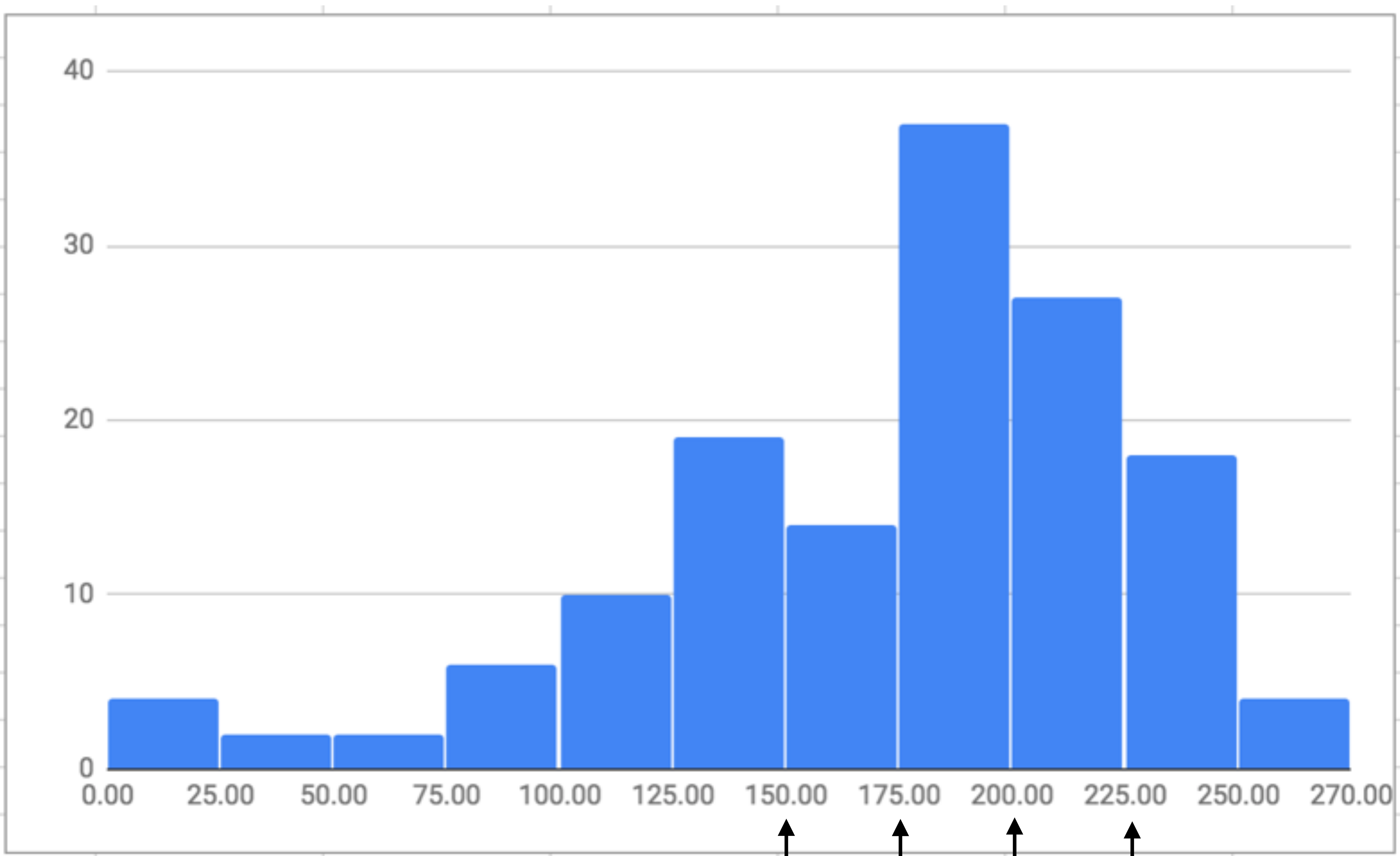
Prof. Zheng Zhang

Rutgers University

November 28, 2018

Class Information

- Project 3 and homework 8 will be released this week.
- Midterm grades are released.



100	86	49	22	143 in total.
D	C	B	A	

Type System

- **Basic types:**
  integer, real, character, …
- **Constructed types:**
  arrays, records (union), sets, pointers, functions
- **A type system is** a collection of rules for assigning type expressions to operators, expressions in the program.
  Type systems are language dependent.
- **A type checker** implements the type system, i.e., deduces type expressions for program constructs based on the type inference rules of the type system.

  The type checker "computes" or "reconstructs" type expressions.

Type Expression

1. A basic type is a type expression. A special basic type, typeError will signal an error. A basic type *void* denotes an untyped statement.
2. Since type expressions may be named, a type name is a type expression. (e.g.: typedef struct foo bar;)
3. Type expressions may contain variables whose values are type expressions (e.g.: useful for languages without type declarations, or polymorphism).
4. A type constructor applied to type expressions is a type expression. Examples:
   (a) arrays
   (b) cartesian products
   (c) records
   (d) pointers
   (e) functions

Example Type Rules

- If both operands of the arithmetic operators of addition, subtraction, and multiplication are of type integer, then the result is of type integer.

Rule for + (analogue rules for − and):

$$\frac{E \vdash e1\text{: } \textit{integer} \qquad E \vdash e2\text{: } \textit{integer}}{E \vdash e1 + e2 : \textit{integer}}$$

- Where E is a type environment that maps constants and variables to their types. In combination with the following axiom in the type system for constants, we can now infer that (2 + 3) is of type integer, E = {2 : integer, 3 : integer}.

In general, type deduction proofs work bottom up.

Example Type Rules

α is a type variable, a placeholder for other type expressions.

- The result of the unary & operator is a pointer to the object referred to by the operand. If the operand is of type “foo”, then the type of the result is a pointer to “foo”. (C and C++ definition)

$$\frac{E \vdash e : \alpha}{E \vdash \&e : \mathit{pointer}(\alpha)}$$

Example Type Rules

α is a type variable, a placeholder for other type expressions.

- Two expressions can only be compared if they have the same types. The result is of type boolean.

$$\frac{E \vdash e1 : \alpha \qquad E \vdash e2 : \alpha}{E \vdash (e1 == e2) : \text{boolean}}$$

Type Variables

Type expressions may contain variables (type variables) whose values are type expressions. **Type variables** are used for implicitly typed languages or languages with polymorphic types.

Programming languages can be

- explicitly typed — every object is declared with its type (**type checking**)
- implicitly typed — type of object is derived from its use (**type reconstruction**)
- monomorphic — every function or data type has a unique, single type
- polymorphic — allows functions or data types to have more than one type (e.g.: *list* in Scheme and *&* in C)

Type Variables — Polymorphic

- **Polymorphic cons:**

$$\frac{E \vdash e1\colon \alpha \qquad E \vdash e2\colon \mathrm{list}(\alpha)}{E \vdash \mathrm{cons}(e1, e2) : \mathrm{list}(\alpha)}$$

- **Polymorphic ‘( ):**

$$E \vdash \text{‘}()\colon \mathrm{list}(\alpha) \;\; \forall\, \alpha \text{ in } E$$

Type Variables: Implicitly Typed

- **Recall:**

$$\frac{E \vdash e1\text{: } \textit{integer} \qquad E \vdash e2\text{: } \textit{integer}}{E \vdash e1 + e2 : \textit{integer}}$$

where E is a type environment. In other words, +" has the type expression (integer×integer) → integer. What are the types of the variables a and b in the following program:

read(a);	{a: α}
read(b);	{a: α, b: β}
... a + b ...;	unify(α, integer)
	unify(β, integer)
	apply type rule; result integer

Unification

unify generates a mapping U from type variables to type expressions such that two type expressions become identical.

Example:

- Two type expressions:

$$\text{type\_expr1} = \alpha \rightarrow \beta$$
$$\text{type\_expr2} = (\beta \times \beta) \rightarrow \text{integer}$$

- Mapping U = unify(type_expr1, type_expr2) implies:

$$\{[\alpha, (\text{integer} \times \text{integer})], [\beta, \text{integer}]\}.$$

- $U(\text{type\_expr}_1) = U(\text{type\_expr}_2)$

$$(\text{integer} \times \text{integer}) \rightarrow \text{integer}$$

Type Inference

Here's an untyped program:

λa.λb.λc. if a (b + 1) then b else c

Informal inference:

- b must be int
- a must be some kind of function
- the argument type of a must be the same as b + 1
- the result type of a must be bool
- the type of c must be the same as b

Putting all these pieces together:

a : int -> bool, b : int, c : int

Find unifier for t1 and t2

If t1 and t2

- are both type variables v1 and v2
  - if v1 = v2, return empty substitution
  - otherwise return {v2:=v1}
- are both primitive types
  - if they are the same, return the empty substitution
  - otherwise, there is no unifier
- both are product types with t1 = (t11*t12) and t2 = (t21*t22)
  - compute the most general unifier S of t11 and t21
  - compute the most general unifier S’ of S t12 and S t22
  - return S ∪ S’
- only one is is type variable v, the other an arbitrary term t
  - if v occurs in t, there is no unifier (occurs check)
  - otherwise, return {v:= t}
- otherwise, there is no unifier

Concurrent Programming Fundamentals

- A THREAD is a potentially-active execution context
- One thread can run concurrently with other threads
- A thread can be thought of as an abstraction of a physical processor
- Classic von Neumann model of computing has single thread of control, while parallel programs have more than one

Concurrent Programming Fundamentals

- Threads can run asynchronously
- The steps of different threads can be interleaved arbitrarily
- Synchronization is a way to ensure that events in different threads happen in a desired order

Thread 1	**Thread 2**
r1 := shared_counter	
r1 := r1 + 1	
shared_counter := r1	r1 := shared_counter
	r1 := r1 + 1
	shared_counter := r1

Concurrent Programming Fundamentals

- Make a sequence of operations **atomic** for shared memory objects
- One possible way to do this is to use locks

```
r1 := shared_counter
r1 := r1 + 1
shared_counter := r1
```

```
acquire(Lock);
r1 := shared_counter
r1 := r1 + 1
shared_counter := r1
release(Lock);
```

A ***lock*** is a construct that, at any point in time, is unowned or owned by a single thread. If a thread *t1* wishes to acquire ownership of a lock that is already owned by another thread *t2*, then *t1* must wait until *t2* releases ownership of the lock.

Compare and Swap (CAS)

- (Intrinsic) atomic instruction available on most processors
- Most common building block for non-blocking algorithms

```
int compare_and_swap(int* reg, int oldval, int newval)
{
        ATOMIC();
        int old_reg_val = *reg;
        if (old_reg_val == oldval)
                *reg = newval;
        END_ATOMIC();
        return old_reg_val;
}
```

- Other types of atomic operations exist:
  increment, decrement, exchange, fetch-and-add, …

Blocking V.S. Non-blocking

- **Blocking algorithms**

If the thread that currently holds the lock is delayed, then all other threads attempting to access the shared data object are also delayed.

- **Non-blocking algorithms**

The delay of a thread does not cause the delay of others. By definition, these algorithms cannot use locks.

```
acquire(Lock)
r1 := shared_counter
r1 := r1 + 1
shared_counter := r1
release(Lock)
```

blocking implementation

```
do
  r1 := shared_counter
while CAS(shared_counter, r1, r1+1)
```

non-blocking implementation

Blocking V.S. Non-blocking

Non-blocking algorithm

The key

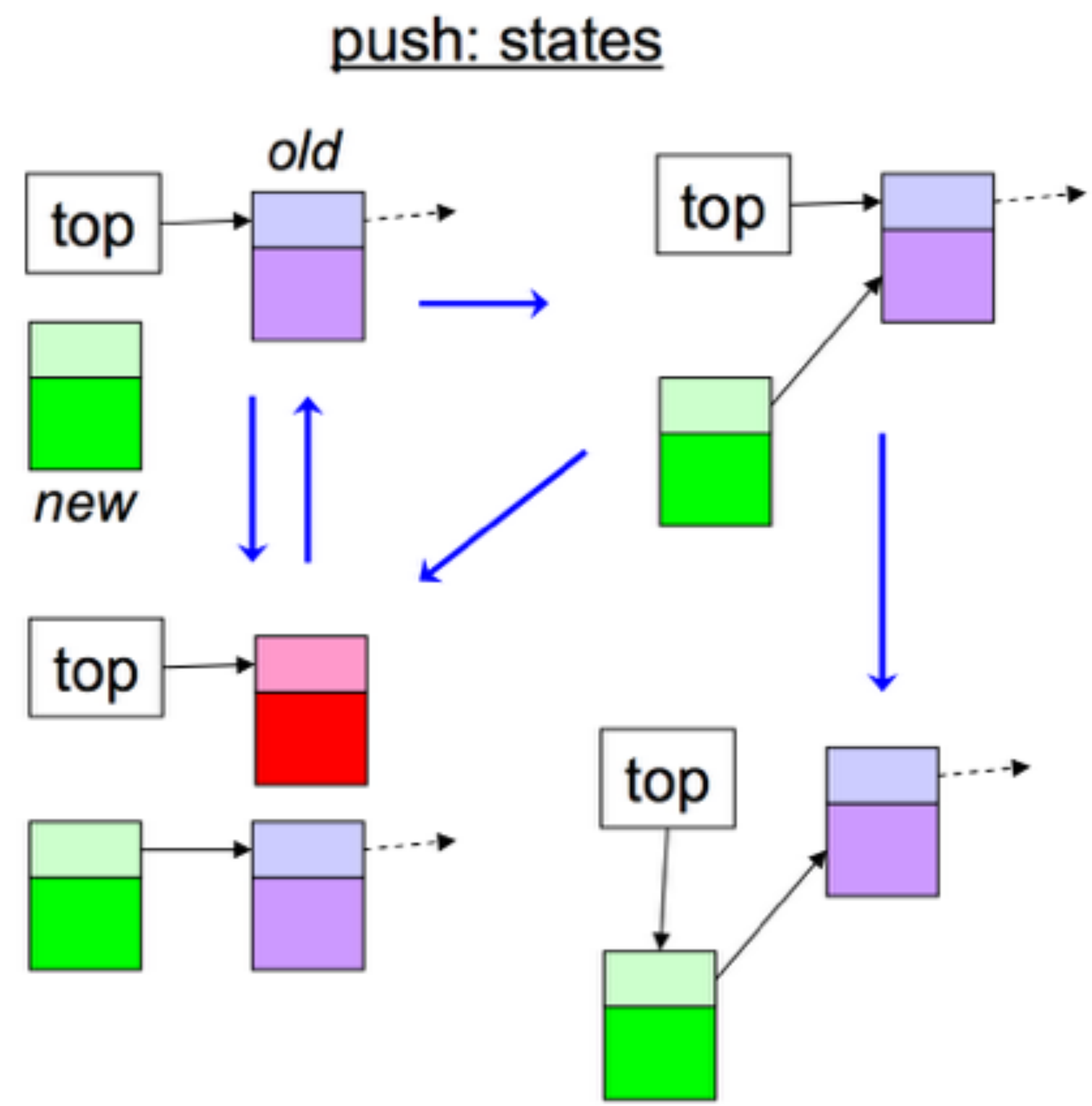
- Try to compute speculatively.
- CAS before committing the result.
- Retry if CAS fails.

Good practice:

- Work with a state-machine.
- Every state must be consistent.
- States = committed (intermediate) results.

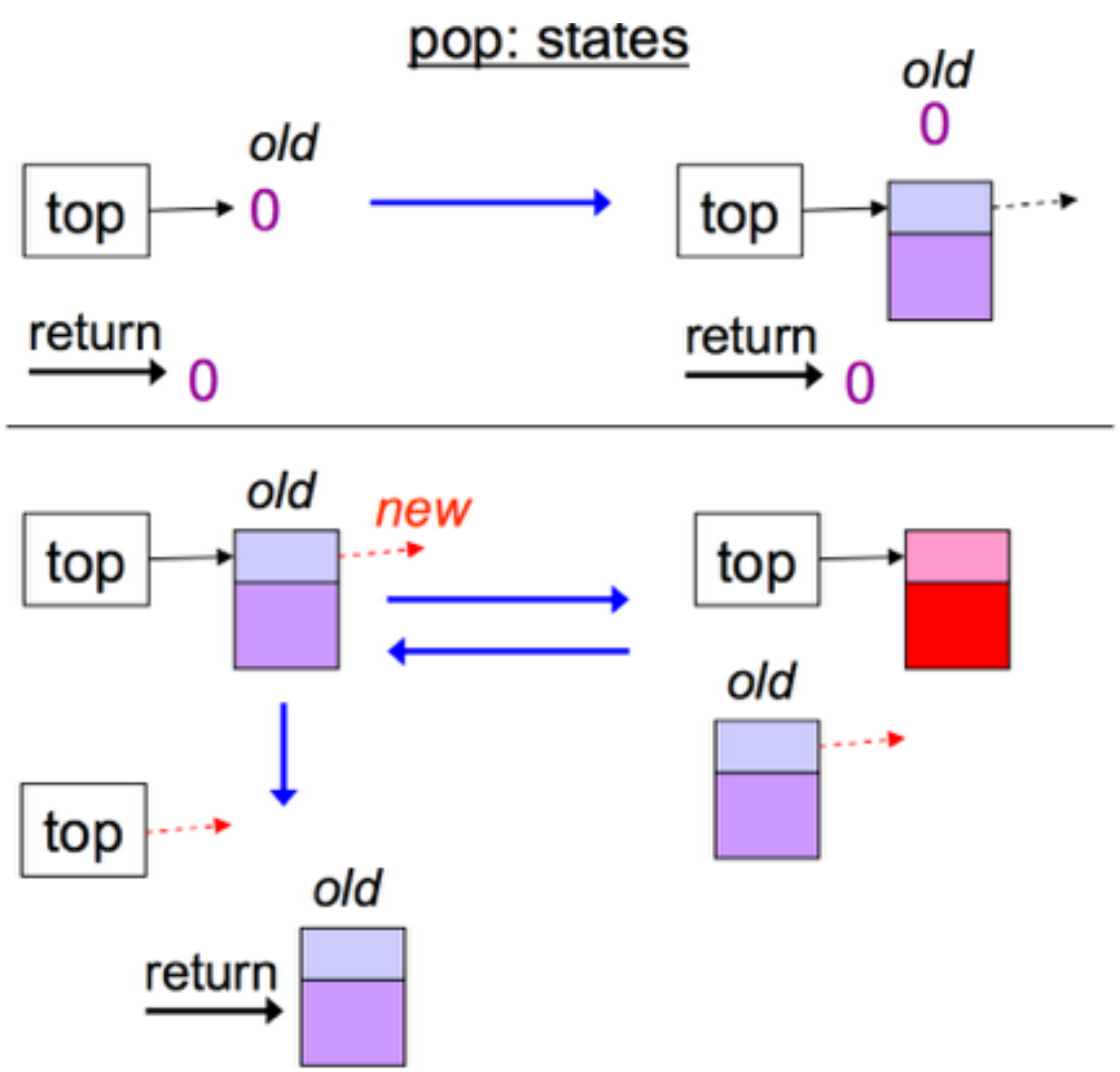
Non-blocking Stack (Treiber's algorithm)

```
proc push(new)
  do
    old = top
    new.next = old
  while not CAS(top, old, new)
end
proc pop
  do
    old = top
    return null if old == null
    new = old.next
  while not CAS(top, old, new)
  return old
end
```



Non-blocking Stack (Treiber's algorithm)

```
proc push(new)
  do
    old = top
    new.next = old
  while not CAS(top, old, new)
end
proc pop
  do
    old = top
    return null if old == null
    new = old.next
  while not CAS(top, old, new)
  return old
end
```



Concurrent Specification

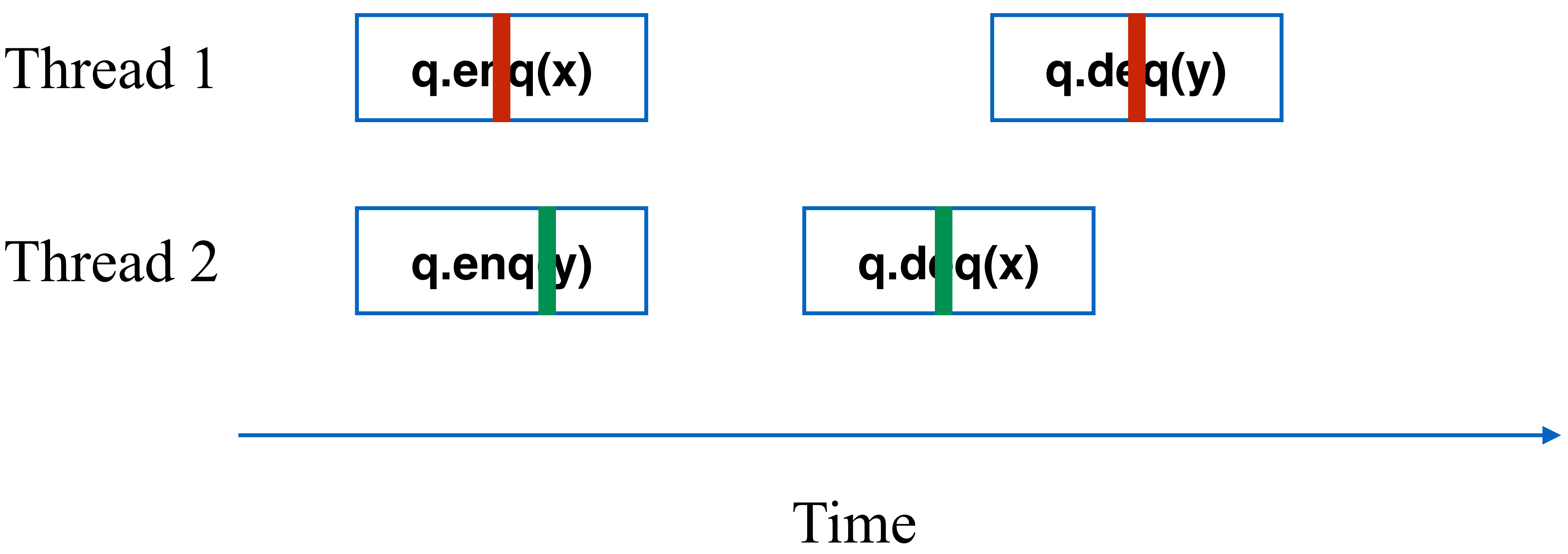
- Given a concurrent data object, each of its methods takes time.
- For example, the **enqueue** and **dequeue** operations for a FIFO queue.
- For a concurrent data object, if we let its methods
  - "take effect"
  - as if instantaneously
  - between invocation and response events

If the corresponding "sequential" behavior can be proved correct, then we say this concurrent data object is linearizable!

Linearizability

- Concurrent FIFO queue examples:

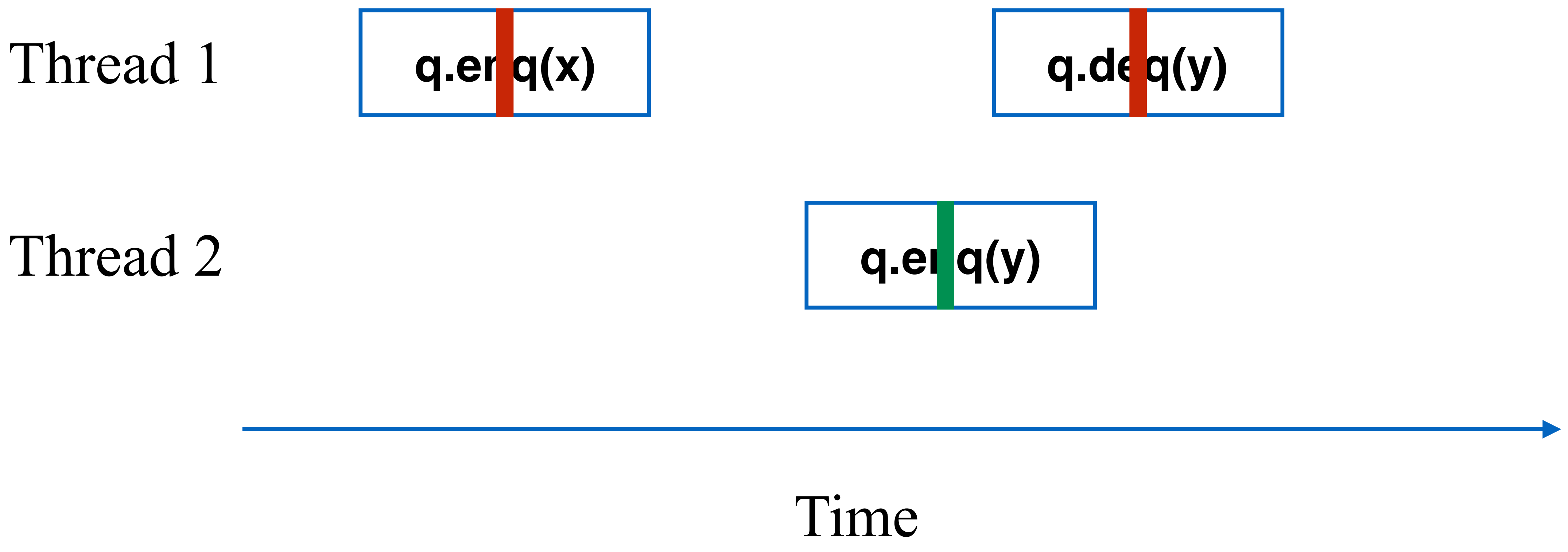
linearizable!



Linearizability

- Concurrent FIFO queue examples:

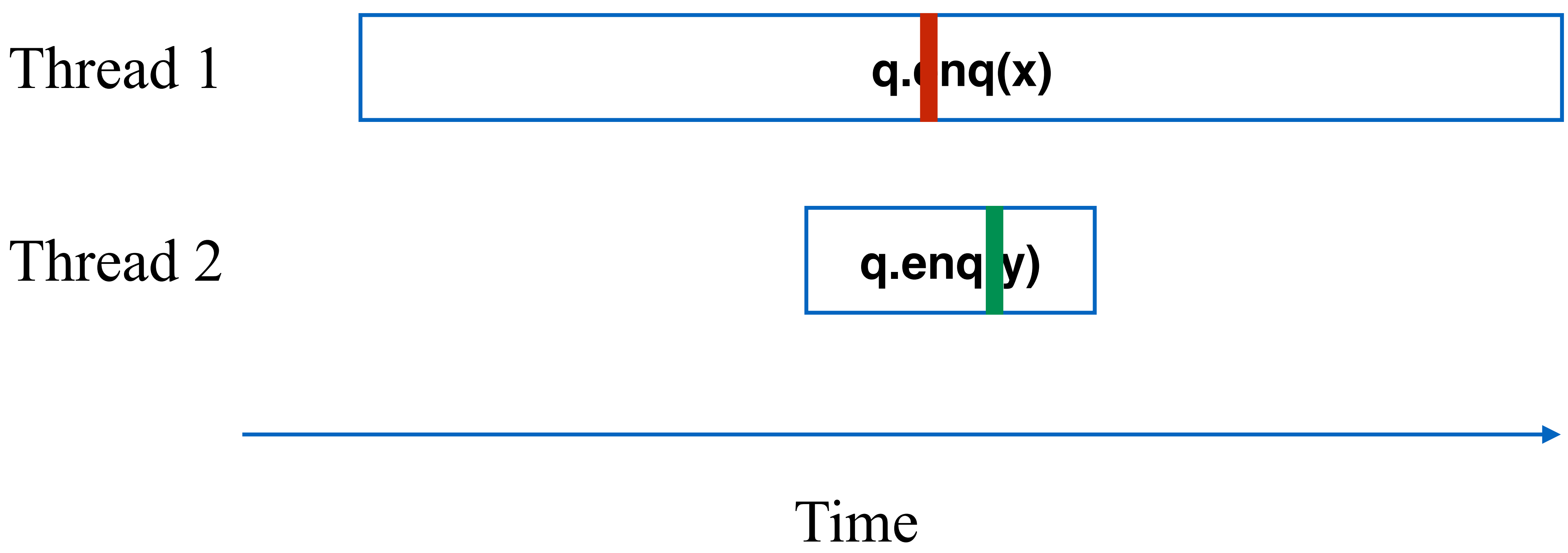
not linearizable!



Linearizability

- Concurrent FIFO queue examples:

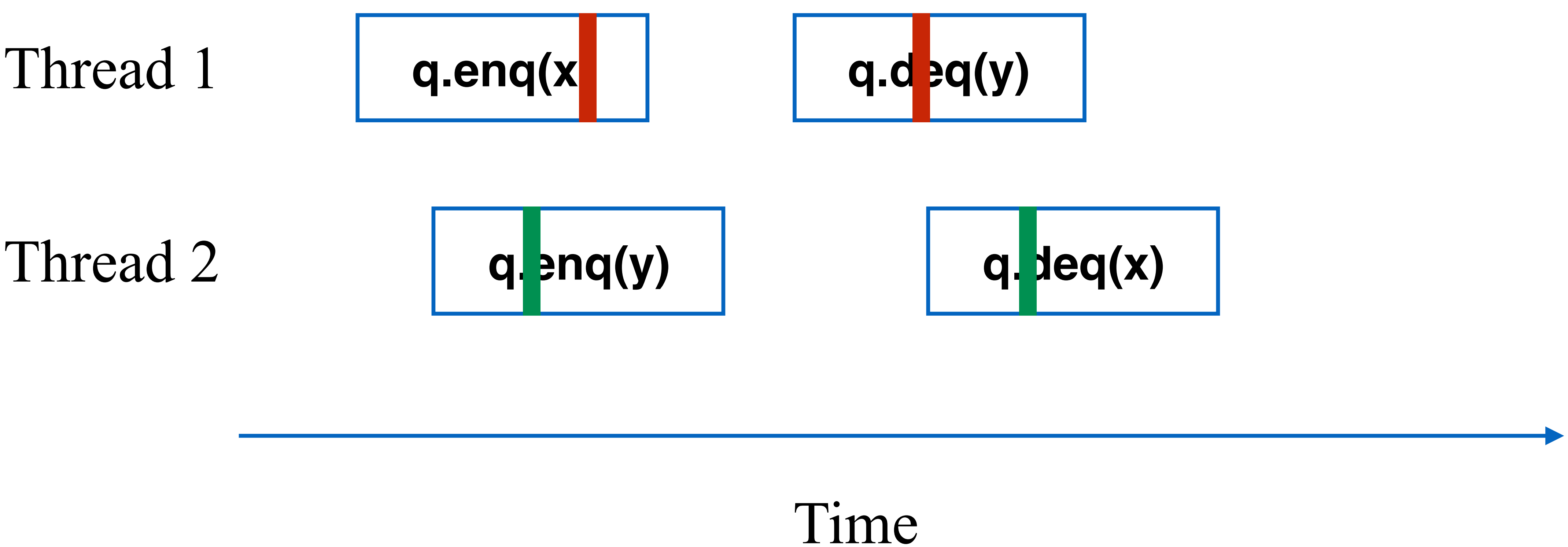
linearizable!



Linearizability

- Concurrent FIFO queue examples:

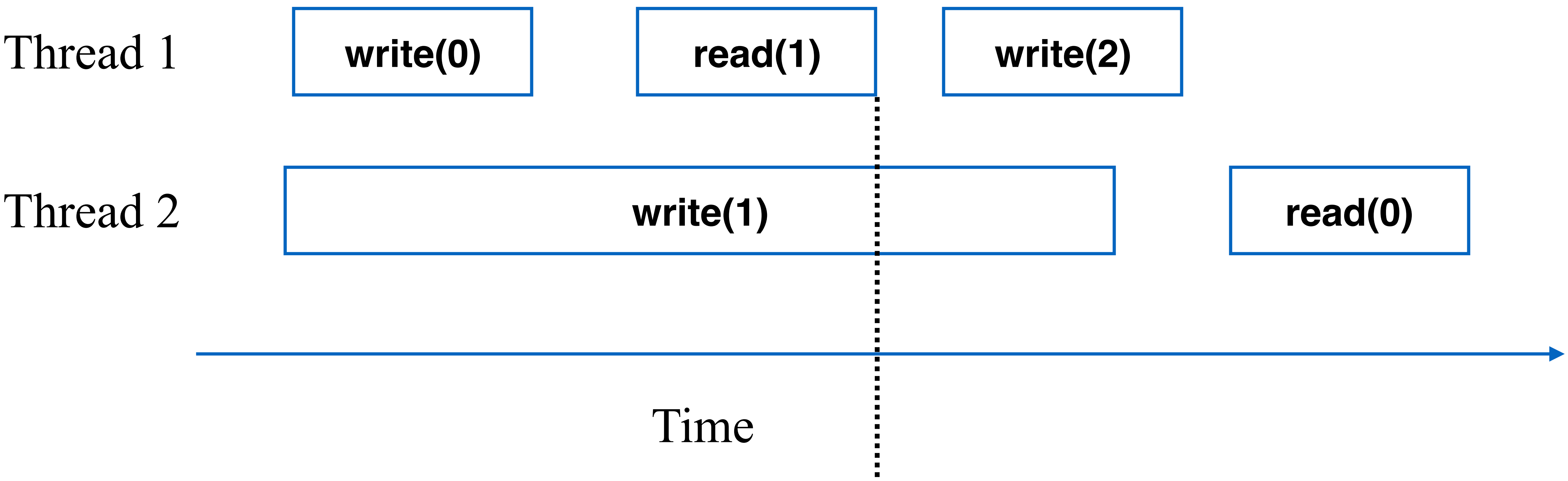
linearizable!



Linearizability

- Read/Write register examples:

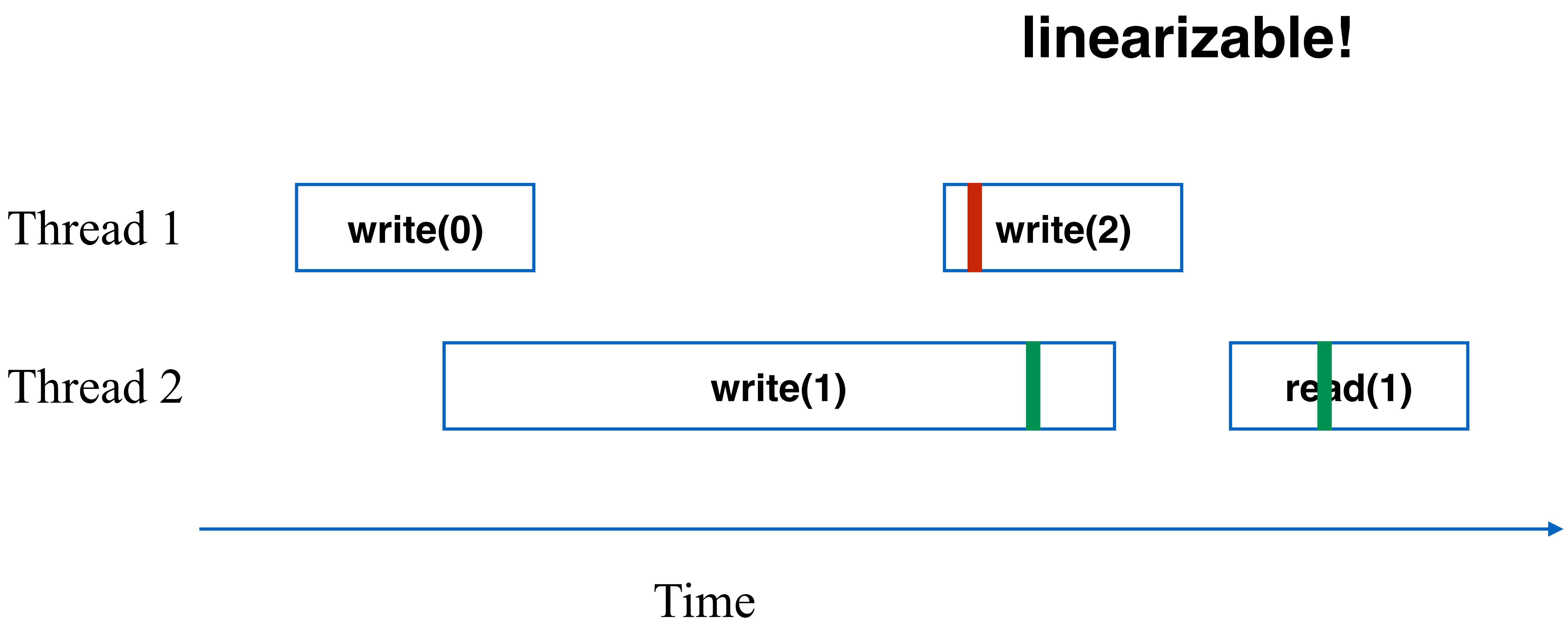
not linearizable!



write(1) already happened!

Linearizability

- Read/Write register examples:



Find Linearization Point

- Easy for data structures that use locks for synchronization: let the linearization point be at where the lock is released
- May not be so easy for those that does not use locks, linearization point might depend on the execution of program
- Nonetheless, linearization analysis is a powerful specification tool for concurrent data objects. It allows us to capture the notion of objects being “atomic” and reason about the correctness.

```
acquire(Lock)
r1 := shared_counter
r1 := r1 + 1
shared_counter := r1
release(Lock)
```

linearization point

Next Class

Reading

- Scott, Chapter 7.2, 13.1 - 13.3;