Transactions / 2 Phase commit
Lock-Freedom
Sequential consistency/Linearizability

Emmett Witchel

CS380L

Problem: Unreliability

- Want reliable update of two resources (e.g. in two disks, machines…)
  - Move file from A to B
  - Create file (update free list, inode, data block)
  - Bank transfer (move $100 from my account to VISA account)
  - Move directory from server A to B
- Machines can crash, messages can be lost

Problem: Unreliability

- Want reliable update of two resources (e.g. in two disks, machines…)
  - Move file from A to B
  - Create file (update free list, inode, data block)
  - Bank transfer (move $100 from my account to VISA account)
  - Move directory from server A to B
- Machines can crash, messages can be lost

Canonical examples:

```
move(file, old-dir, new-dir) {
    delete(file, old-dir)
    add(file, new-dir)
}
```

```
create(file, dir) {
    alloc-disk(file, header, data)
    write(header)
    add (file, dir)
}
```

Problem: Unreliability

Can we use messages? E.g. with retries over unreliable medium to synchronize with guarantees?

- Want reliable update of two resources (e.g. in two
  - Move file from A to B
  - Create file (update free list, inode, data block)
  - Bank transfer (move $100 from my account to VISA account)
  - Move directory from server A to B
- Machines can crash, messages can be lost

Canonical examples:

```
move(file, old-dir, new-dir) {
    delete(file, old-dir)
    add(file, new-dir)
}
```

```
create(file, dir) {
    alloc-disk(file, header, data)
    write(header)
    add (file, dir)
}
```

Problem: Unreliability

- Want reliable update of two resources (e.g. in two
  - Move file from A to B
  - Create file (update free list, inode, data block)
  - Bank transfer (move $100 from my account to VISA acc
  - Move directory from server A to B
- Machines can crash, messages can be lost

Can we use messages? E.g. with retries over unreliable medium to synchronize with guarantees?

No.
Not even if all messages get through!

Canonical examples:

```
move(file, old-dir, new-dir) {
    delete(file, old-dir)
    add(file, new-dir)
}
```

```
create(file, dir) {
    alloc-disk(file, header, data)
    write(header)
    add (file, dir)
}
```

Problem: Unreliability

- Want reliable update of two resources (e.g. in two
  - Move file from A to B
  - Create file (update free list, inode, data block)
  - Bank transfer (move $100 from my account to VISA acc
  - Move directory from server A to B
- Machines can crash, messages can be lost

Canonical examples:

```
move(file, old-dir, new-dir) {
    delete(file, old-dir)
    add(file, new-dir)
}
```

```
create(file, dir)
    alloc-disk(file
    write(header)
    add (file, dir)
}
```

Can we use messages? E.g. with retries over unreliable medium to synchronize with guarantees?

No.
Not even if all messages get through!

- Transactions: solve weaker problem:
  - 2 things will either happen or not
  - not necessarily at the same time
- Core idea
  - one entity: yes or no for all

Transactional Programming Model

```
begin transaction;
    x = read("x-values", ....);
    y = read("y-values", ....);
    z = x+y;
    write("z-values", z, ....);
commit transaction;
```

Review: ACID Semantics

- Atomic – all updates happen or none do
- Consistent – system invariants maintained across updates
- Isolated – no visibility into partial updates
- Durable – once done, stays done
- Are subsets ever appropriate?

```
begin transaction;
    x = read(“x-values”, ....);
    y = read(“y-values”, ....);
    z = x+y;
    write(“z-values”, z, ....);
commit transaction;
```

Transactions: Implementation

- Key idea: turn multiple updates into a single one
- Many implementation Techniques
  - Two-phase locking
  - Timestamp ordering
  - Optimistic Concurrency Control
  - Journaling
  - 2,3-phase commit
  - Speculation-rollback
  - Single global lock
  - Compensating transactions

Transactions: Implementation

- Key idea: turn multiple updates into a single one
- Many implementation Techniques
  - Two-phase locking
  - Timestamp ordering
  - Optimistic Concurrency Control
  - Journaling
  - 2,3-phase commit
  - Speculation-rollback
  - Single global lock
  - Compensating transactions

Key problems:

- output commit
- synchronization

Transactions: Implementation

- Key idea: turn multiple updates into a single one
- Many implementation Techniques
  - Two-phase locking
  - Timestamp ordering
  - Optimistic Concurrency Control
  - Journaling
  - 2,3-phase commit
  - Speculation-rollback
  - Single global lock
  - Compensating transactions

Key problems:

- output commit
- synchronization

Implementing Transactions

```
BEGIN_TXN();
    x = read(“x-values”, ....);
    y = read(“y-values”, ....);
    z = x+y;
    write(“z-values”, z, ....);
COMMIT_TXN();
```

Implementing Transactions

```
BEGIN_TXN();
    x = read("x-values", ....);
    y = read("y-values", ....);
    z = x+y;
    write("z-values", z, ....);
COMMIT_TXN();
```

```
BEGIN_TXN() {

}
```

```
COMMIT_TXN() {

}
```

Implementing Transactions

```
BEGIN_TXN();
    x = read("x-values", ....);
    y = read("y-values", ....);
    z = x+y;
    write("z-values", z, ....);
COMMIT_TXN();
```

```
BEGIN_TXN() {
    LOCK(single-global-lock);
}
```

```
COMMIT_TXN() {
    UNLOCK(single-global-lock);
}
```

Implementing Transactions

```
BEGIN_TXN();
    x = read("x-values", ....);
    y = read("y-values", ....);
    z = x+y;
    write("z-values", z, ....);
COMMIT_TXN();
```

```
BEGIN_TXN() {
    LOCK(single-global-lock);
}
```

```
COMMIT_TXN() {
    UNLOCK(single-global-lock);
}
```

Pros/Cons?

Review: Two-phase locking

- Phase 1 (acquire): only acquire locks in order
- Phase 2: unlock (after first unlock, no more locks)
- +avoids deadlock
- - can hold locks longer than necessary

```
Lock x, y
x = x + 1
y = y – 1
unlock y, x
```

```
Lock x
x = x + 1
Lock y
y = y – 1
unlock y, x
```

Using two-phase locking for transactions

```
BEGIN_TXN();
x = x + 1
y = y – 1
COMMIT_TXN();
```

Using two-phase locking for transactions

```
BEGIN_TXN();
x = x + 1
y = y - 1
COMMIT_TXN();
```

```
BEGIN_TXN() {

}
```

```
COMMIT_TXN() {

}
```

Using two-phase locking for transactions

```
BEGIN_TXN();
x = x + 1
y = y - 1
COMMIT_TXN();
```

```
BEGIN_TXN() {
  rwset = Union(rset, wset);
  rwset = sort(rwset);
  forall x in rwset
    LOCK(x);
}
```

```
COMMIT_TXN() {
  forall x in rwset
    UNLOCK(x);
}
```

Using two-phase locking for transactions

```
BEGIN_TXN();
x = x + 1
y = y - 1
COMMIT_TXN();
```

```
BEGIN_TXN() {
  rwset = Union(rset, wset);
  rwset = sort(rwset);
  forall x in rwset
     LOCK(x);
}
```

```
COMMIT_TXN() {
  forall x in rwset
     UNLOCK(x);
}
```

Pros/Cons?

Using two-phase locking for transactions

```
BEGIN_TXN();
x = x + 1
y = y - 1
COMMIT_TXN();
```

```
BEGIN_TXN() {
  rwset = Union(rset, wset);
  rwset = sort(rwset);
  forall x in rwset
    LOCK(x);
}
```

```
COMMIT_TXN() {
  forall x in rwset
    UNLOCK(x);
}
```

Pros/Cons?

What happens on failures?

Two-phase commit (distributed transactions)

- N participants agree or don’t (atomicity)
- Phase 1: everyone “prepares”
- Phase 2: Master decides and tells everyone to actually commit
- What if the master crashes in the middle?

Review: 2PC

Phase 1

1. Coordinator sends REQUEST to all participants
2. Participants receive request and
3. Execute locally
4. Write VOTE_COMMIT or VOTE_ABORT to local log
5. Send VOTE_COMMIT or VOTE_ABORT to coordinator

Phase 2

- Case 1: receive VOTE_ABORT or timeout
  - Write GLOBAL_ABORT to log
  - send GLOBAL_ABORT to participants
- Case 2: receive VOTE_COMMIT from all
  - Write GLOBAL_COMMIT to log
  - send GLOBAL_COMMIT to participants
- Participants receive decision, write GLOBAL_* to log

Example—move: C→S1: delete foo from /,
C→S2: add foo to /

Failure case:
S1 writes rm /foo, VOTE_COMMIT to log
S1 sends VOTE_COMMIT
S2 decides permission problem
S2 writes/sends VOTE_ABORT

Success case:
S1 writes rm /foo, VOTE_COMMIT to log
S1 sends VOTE_COMMIT
S2 writes add foo to /
S2 writes/sends VOTE_COMMIT

2PC corner cases

Phase 1

1. Coordinator sends REQUEST to all participants
X 2. Participants receive request and
3. Execute locally
4. Write VOTE_COMMIT or VOTE_ABORT to local log
5. Send VOTE_COMMIT or VOTE_ABORT to coordinator

Phase 2

- Y Case 1: receive VOTE_ABORT or timeout
  - Write GLOBAL_ABORT to log
  - send GLOBAL_ABORT to participants
- Case 2: receive VOTE_COMMIT from all
  - W Write GLOBAL_COMMIT to log
  - send GLOBAL_COMMIT to participants
- Z Participants recv decision, write GLOBAL_* to log

- What if participant crashes at X?
- Coordinator crashes at Y?
- Participant crashes at Z?
- Coordinator crashes at W?

2PC limitation(s)

- Coordinator crashes at W, never wakes up
- All nodes block forever!
- Can participants ask each other what happened?
- 2PC: always has risk of indefinite blocking
- Solution: (yes) 3 phase commit!
  - Reliable replacement of crashed “leader”
  - 2PC often good enough in practice

Problems with locks (pessimistic sync)

Locks: a litany of problems

- Deadlock
- Priority inversion
- Convoys
- Fault Isolation
- Preemption Tolerance
- Performance

Solution: don't use locks

Non-Blocking Synchronization

- Lock-free → Subset of a broader: Non-blocking Synchronization
- Thread-safe access shared mutable state without mutual exclusion
- Possible without HW support
  - E.g. Lamport's Concurrent Buffer
  - But not really practical
- Built on atomic instructions like CAS + clever algorithmic tricks
- Lock-free *algorithms* are hard, so
- General approach: encapsulate lock-free algorithms in data structures
  - Queue, list, hash-table, skip list, etc.
  - New LF data structure → research result

Example: List Append

```
struct Node
{
  int data;
  struct Node *next;
};
```

```
void append(Node** head_ref, int new_data) {
    Node* new_node = mknode(new_data, head_ref);
    lock();
    if (*head_ref == NULL) {
       *head_ref = new_node;
    } else {
       while (last->next != NULL)
           last = last->next;
       last->next = new_node;
    }
    unlock();
}
```

Example: List Append

```
struct Node
{
  int data;
  struct Node *next;
};
```

```
void append(Node** head_ref, int new_data) {
    Node* new_node = mknode(new_data, head_ref);
    lock();
    if (*head_ref == NULL) {
        *head_ref = new_node;
    } else {
        while (last->next != NULL)
            last = last->next;
        last->next = new_node;
    }
    unlock();
}
```

Example: List Append

```
struct Node
{
  int data;
  struct Node *next;
};
```

```
void append(Node** head_ref, int new_data) {
    Node* new_node = mknode(new_data, head_ref);
    lock();
    if (*head_ref == NULL) {
       *head_ref = new_node;
    } else {
       while (last->next != NULL)
           last = last->next;
       last->next = new_node;
    }
    unlock();
}
```

Example: List Append

```
struct Node
{
  int data;
  struct Node *next;
};
```

```
void append(Node** head_ref, int new_data) {
    Node* new_node = mknode(new_data, head_ref);
    lock();
    if (*head_ref == NULL) {
        *head_ref = new_node;
    } else {
        while (last->next != NULL)
            last = last->next;
        last->next = new_node;
    }
    unlock();
}
```

- What property do the locks enforce?

Example: List Append

```
struct Node
{
  int data;
  struct Node *next;
};
```

```
void append(Node** head_ref, int new_data) {
    Node* new_node = mknode(new_data, head_ref);
    lock();
    if (*head_ref == NULL) {
        *head_ref = new_node;
    } else {
        while (last->next != NULL)
            last = last->next;
        last->next = new_node;
    }
    unlock();
}
```

- What property do the locks enforce?
- What does the mutual exclusion ensure?

Example: List Append

```
struct Node
{
  int data;
  struct Node *next;
};
```

```
void append(Node** head_ref, int new_data) {
    Node* new_node = mknode(new_data, head_ref);
    lock();
    if (*head_ref == NULL) {
        *head_ref = new_node;
    } else {
        while (last->next != NULL)
            last = last->next;
        last->next = new_node;
    }
    unlock();
}
```

- What property do the locks enforce?
- What does the mutual exclusion ensure?
- Can we ensure consistent view (invariants hold) sans mutual exclusion?

Example: List Append

```
struct Node
{
  int data;
  struct Node *next;
};
```

```
void append(Node** head_ref, int new_data) {
    Node* new_node = mknode(new_data, head_ref);
    lock();
    if (*head_ref == NULL) {
        *head_ref = new_node;
    } else {
        while (last->next != NULL)
            last = last->next;
        last->next = new_node;
    }
    unlock();
}
```

- What property do the locks enforce?
- What does the mutual exclusion ensure?
- Can we ensure consistent view (invariants hold) sans mutual exclusion?
- Key insight: allow inconsistent view and fix it up algorithmically

Lock-Free Stack

```
struct Node
{
  int data;
  struct Node *next;
};
```

```
void push(int t) {
    Node* node = new Node(t);
    do {
        node->next = head;
    } while (!cas(&head, node, node->next));
}
```

```
bool pop(int& t) {
   Node* current = head;
   while(current) {
       if(cas(&head, current->next, current)) {
           t = current->data;
           return true;
       }
       current = head;
       }
   return false;
}
```

Lock-Free Stack

```
struct Node
{
  int data;
  struct Node *next;
};
```

```
void push(int t) {
    Node* node = new Node(t);
    do {
        node->next = head;
    } while (!cas(&head, node, node->next));
}

bool pop(int& t) {
   Node* current = head;
   while(current) {
       if(cas(&head, current->next, current)) {
          t = current->data;
          return true;
       }
       current = head;
   }
   return false;
}
```

- Why does is it work?

Lock-Free Stack

```
struct Node
{
  int data;
  struct Node *next;
};
```

```
void push(int t) {
    Node* node = new Node(t);
    do {
        node->next = head;
    } while (!cas(&head, node, node->next));
}
```

```
bool pop(int& t) {
    Node* current = head;
    while(current) {
        if(cas(&head, current->next, current)) {
            t = current->data; // problem?
            return true;
        }
        current = head;
    }
    return false;
}
```

- Why does is it work?

Lock-Free Stack

```
struct Node
{
  int data;
  struct Node *next;
};
```

```
void push(int t) {
    Node* node = new Node(t);
    do {
        node->next = head;
    } while (!cas(&head, node, node->next));
}
```

```
bool pop(int& t) {
   Node* current = head;
   while(current) {
       if(cas(&head, current->next, current)) {
           t = current->data; // problem?
           return true;
       }
       current = head;
   }
   return false;
}
```

- Why does is it work?
- Does it enforce all invariants?

Sequential Specification

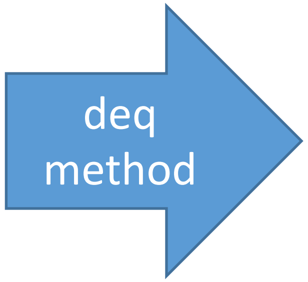
We use pre-conditions and post-conditions.

- **Pre-condition** defines the state of the object before method.
- **Post-condition** defines the state of the object after the method. Also defines returned value and thrown exception.

Sequential Specification

We use pre-conditions and post-conditions.

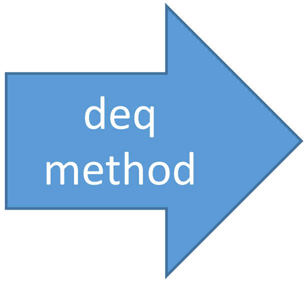
- **Pre-condition** defines the state of the object before method.
- **Post-condition** defines the state of the object after the method. Also defines returned value and thrown exception.



Sequential Specification

We use pre-conditions and post-conditions.

- **Pre-condition** defines the state of the object before method.
- **Post-condition** defines the state of the object after the method. Also defines returned value and thrown exception.



Pre-condition:

queue is not empty.

Post-condition:

- Returns first item in queue.
- Removes first item in queue.

Sequential Specification

We use pre-conditions and post-conditions.

- **Pre-condition** defines the state of the object before method.
- **Post-condition** defines the state of the object after the method. Also defines returned value and thrown exception.

deq method

Pre-condition:

queue is not empty.

Post-condition:

- Returns first item in queue.
- Removes first item in queue.

Pre-condition:

queue is empty.

Post-condition:

- Throws EmptyException.
- Queue state is unchanged.

Sequential Specification

We use pre-conditions and post-conditions.

- **Pre-condition** defines the state of the object before method.
- **Post-condition** defines the state of the object after the method. Also defines returned value and thrown exception.

Concurrent Specifications

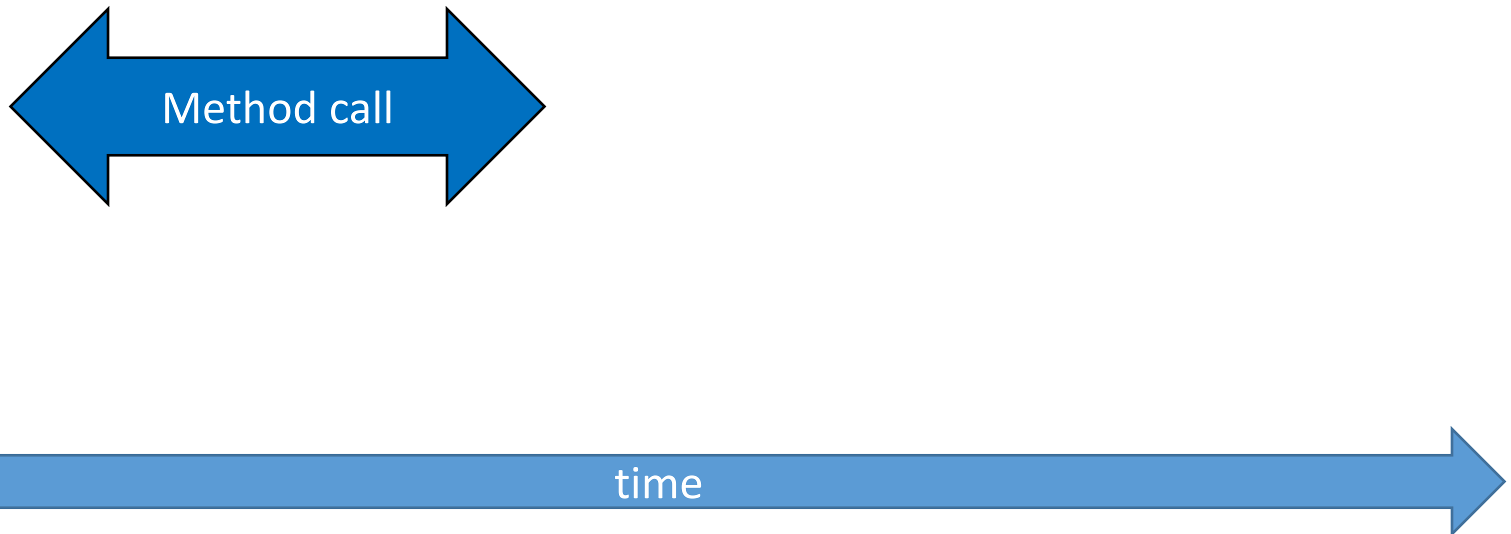
- Methods here “take time”.
- In sequential computing, methods take time also, but we don’t care.
  - In sequential: method call is an event.
  - In concurrent: method call is an interval.
- Methods intervals can overlap

Concurrent Specifications

- Methods here “take time”.
- In sequential computing, methods take time also, but we don’t care.
  - In sequential: method call is an event.
  - In concurrent: method call is an interval.
- Methods intervals can overlap

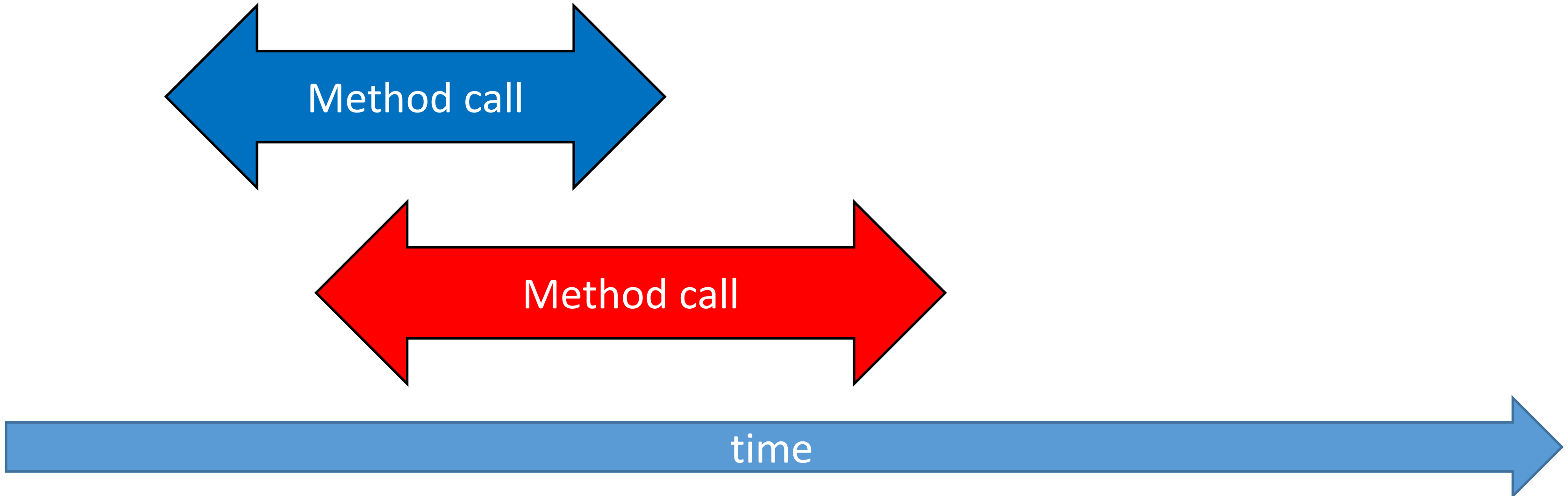
Concurrent Specifications

- Methods here “take time”.
- In sequential computing, methods take time also, but we don’t care.
  - In sequential: method call is an event.
  - In concurrent: method call is an interval.
- Methods intervals can overlap



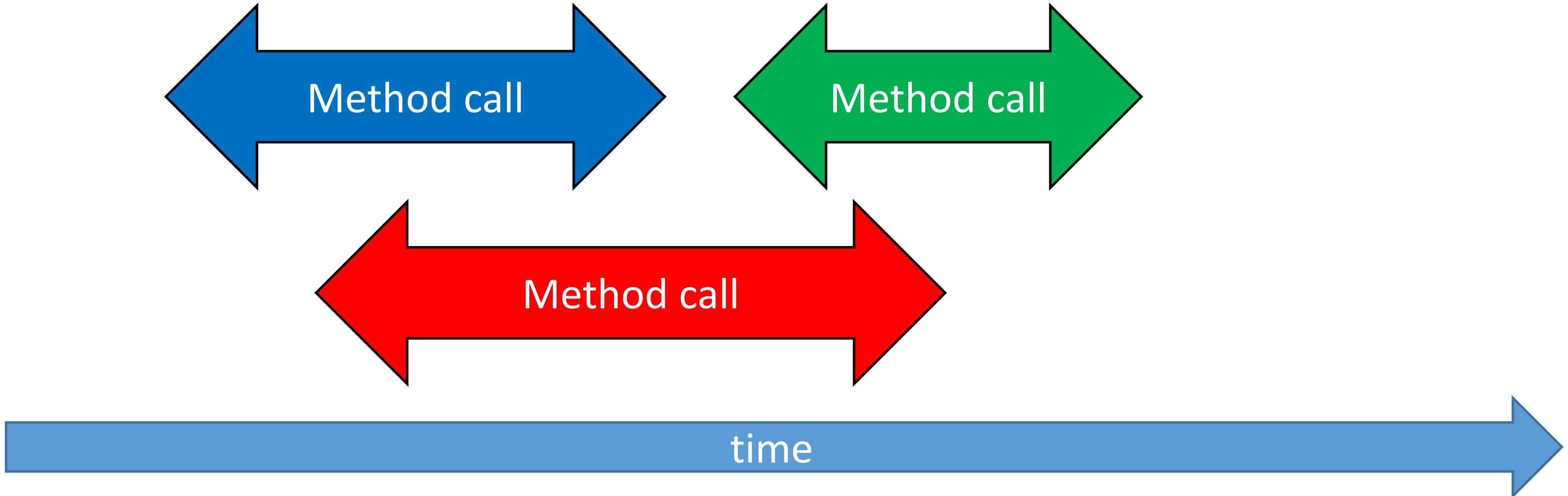
Concurrent Specifications

- Methods here “take time”.
- In sequential computing, methods take time also, but we don’t care.
  - In sequential: method call is an event.
  - In concurrent: method call is an interval.
- Methods intervals can overlap



Concurrent Specifications

- Methods here “take time”.
- In sequential computing, methods take time also, but we don’t care.
  - In sequential: method call is an event.
  - In concurrent: method call is an interval.
- Methods intervals can overlap



Concurrent objects

- An object in languages such as Java and C++ is a container for data.
    - Methods are the only way to access state
- Each object has a class which describes how its methods behave.
    - Can have a list that allows append only, or allows insert. Different APIs
- Given object, is its behavior correct during concurrent execution?
    - Sequential consistency – good for some things, but weak
        - Respects program order
    - Linearizability
        - Composable: If all objects are linearlizable, system is linearizable
        - Core idea: each ***operation***
        - 1. takes effect instantaneously
        - 2. at some point between its invocation and its response.

Correctness criteria

Look at the behaviour of the data structure

- what operations are called on it
- what their results are

If behaviour is indistinguishable from atomic calls to a sequential implementation then the concurrent implementation is correct.

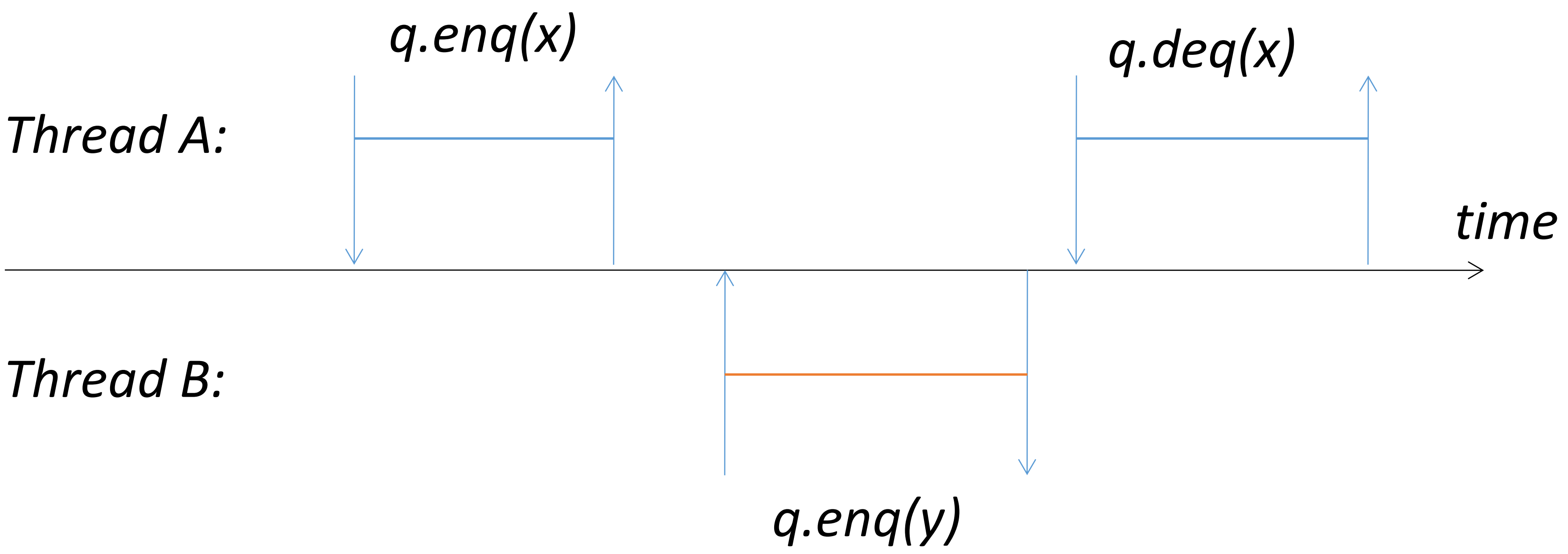
Sequential consistency definition

- Method calls should appear to happen in a one-at-a-time, sequential order
- Method calls should appear to take effect in program order.

- NB: Says nothing of ordering of methods from different threads/programs.

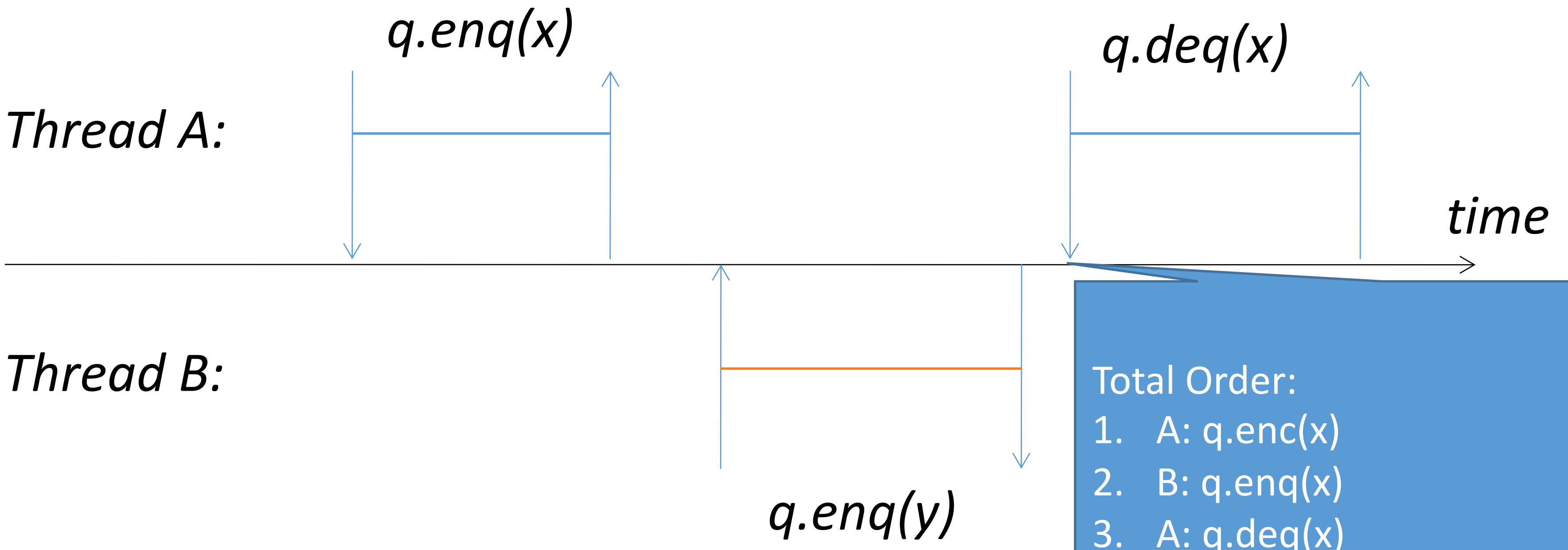
Sequential consistency example

q is a FIFO queue



Sequential consistency example

q is a FIFO queue



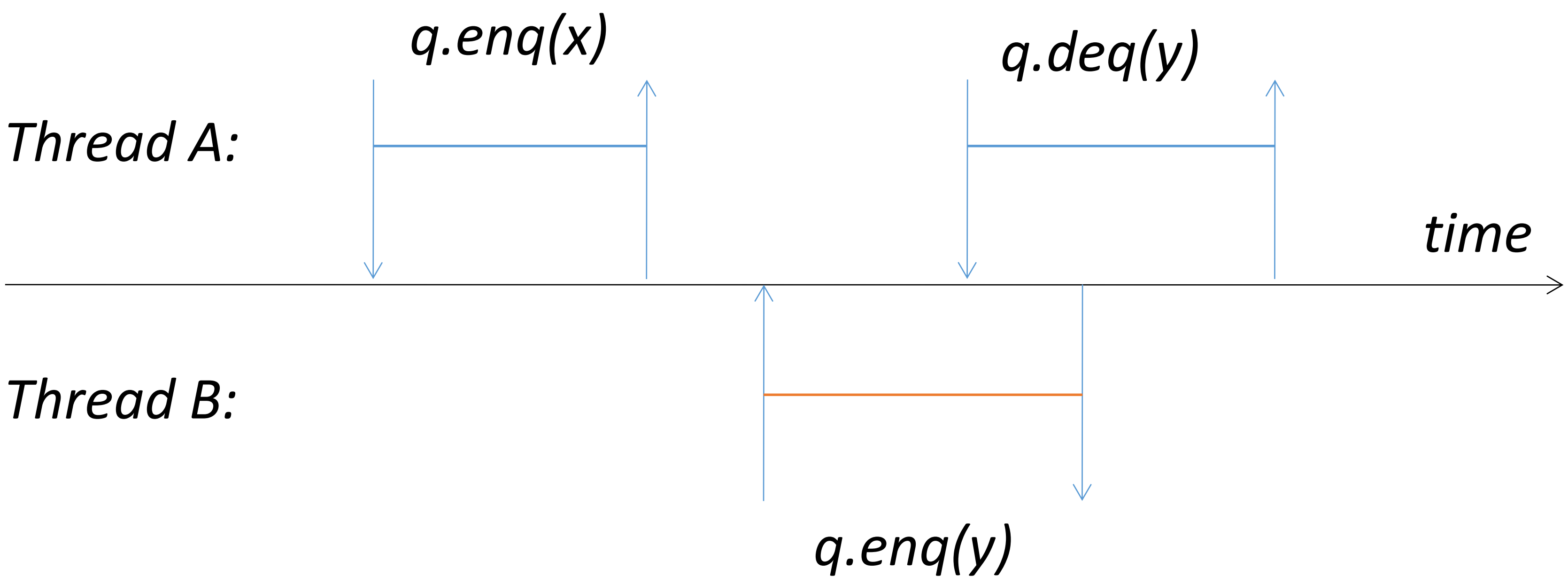
Total Order:

1. A: q.enc(x)
2. B: q.enq(x)
3. A: q.deq(x)

- Total order follows real-time order

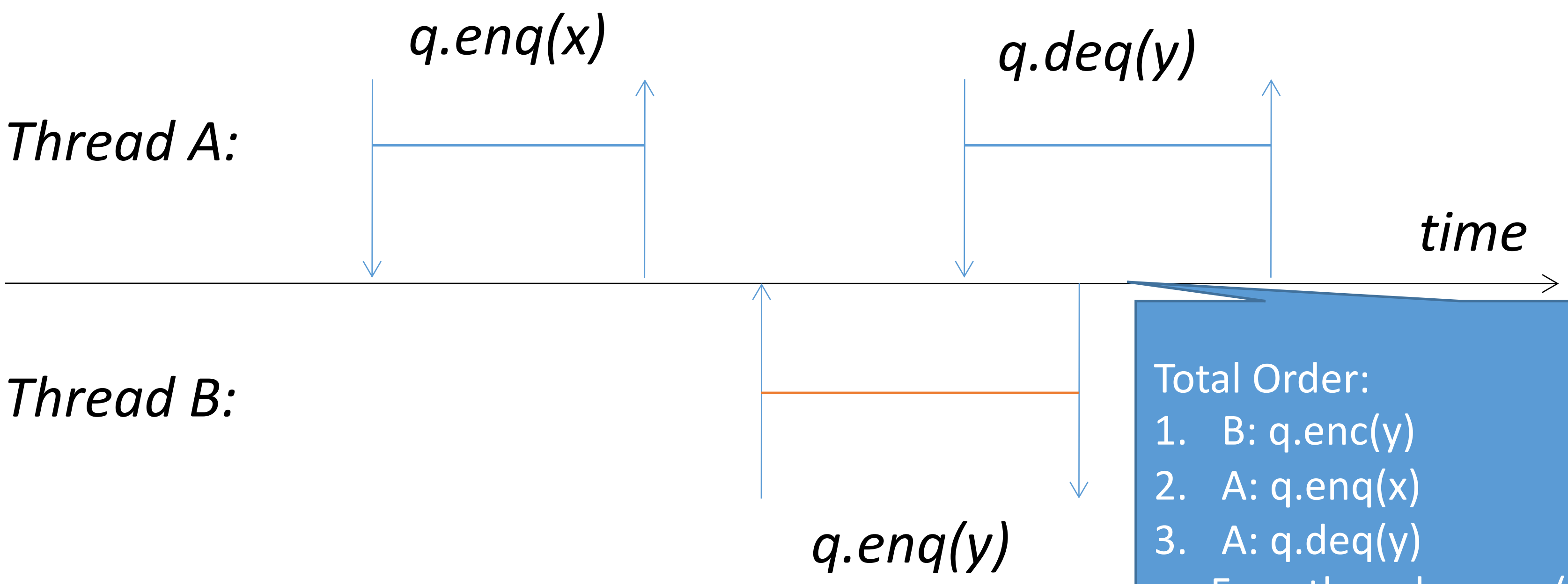
Sequential consistency more complexity

q is a FIFO queue



Sequential consistency more complexity

q is a FIFO queue



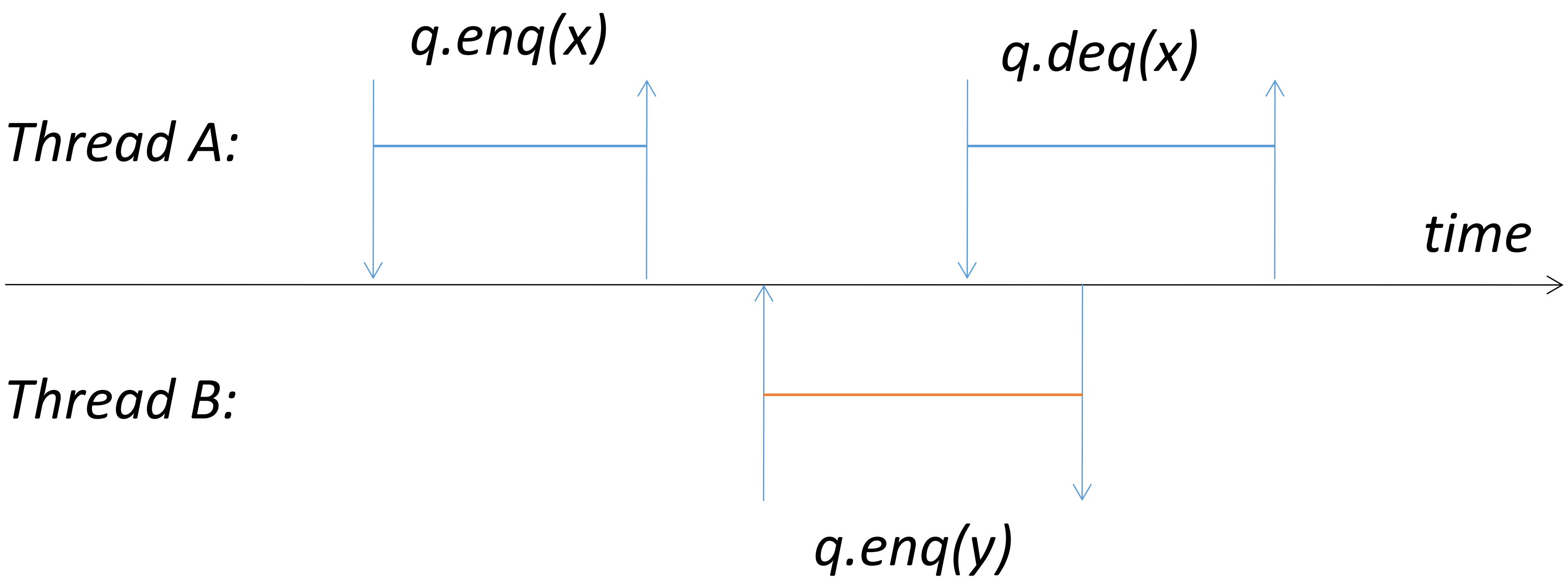
Total Order:
1. B: q.enc(y)
2. A: q.enq(x)
3. A: q.deq(y)
- Even though q.enq(y) starts in real time after q.enq(x) finishes

Linearizable definition

- Method calls should appear to happen in a one-at-a-time, sequential order
- ~~Method calls should appear to take effect in program order.~~
- Each method call should appear to take effect instantaneously at some moment between its invocation and response.
  - Often called its linearization point

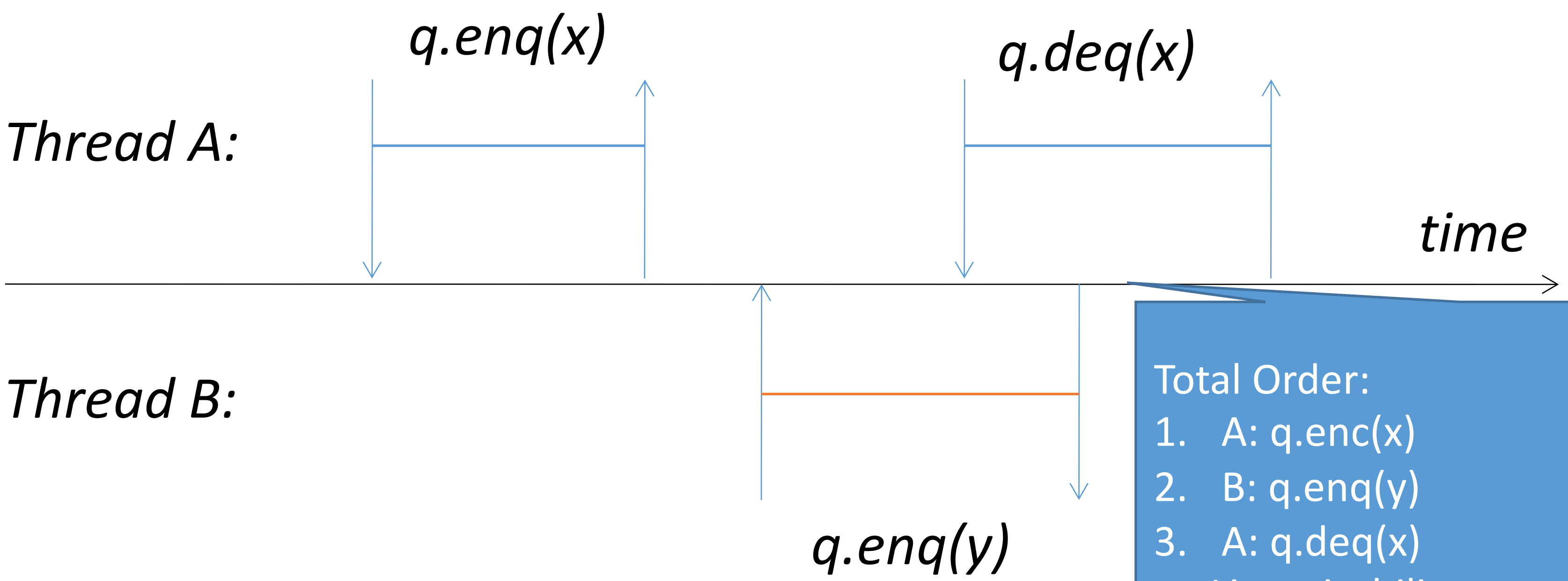
Linearizability

q is a FIFO queue



Linearizability

q is a FIFO queue



Total Order:

1. A: q.enc(x)
2. B: q.enq(y)
3. A: q.deq(x)

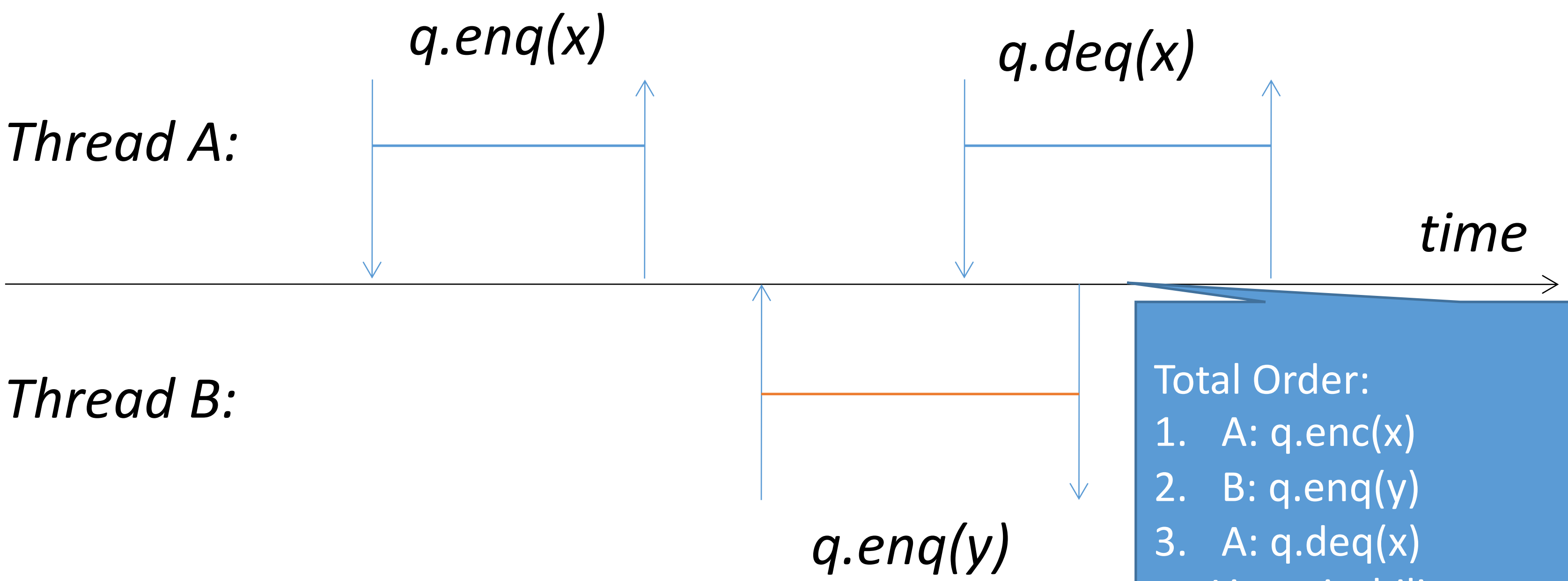
- Linearizability respects real time order

Linearizability

q is a FIFO queue

Linearizability:

- Is there a correct sequential history:
  - Same results as the concurrent one
  - Consistent with the timing of the invocations/responses?
  - Start/end impose ordering constraints



Total Order:

1. A: q.enc(x)
2. B: q.enq(y)
3. A: q.deq(x)

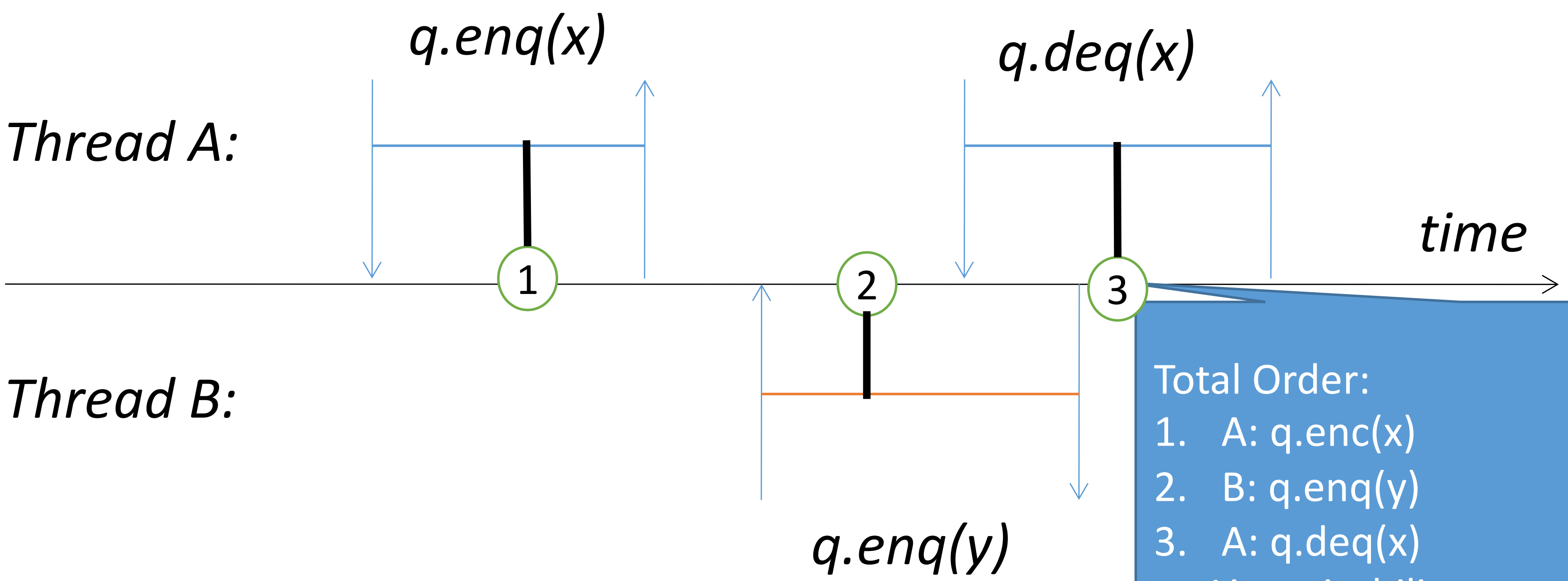
- Linearizability respects real time order

Linearizability

q is a FIFO queue

Linearizability:

- Is there a correct sequential history:
  - Same results as the concurrent one
  - Consistent with the timing of the invocations/responses?
  - Start/end impose ordering constraints



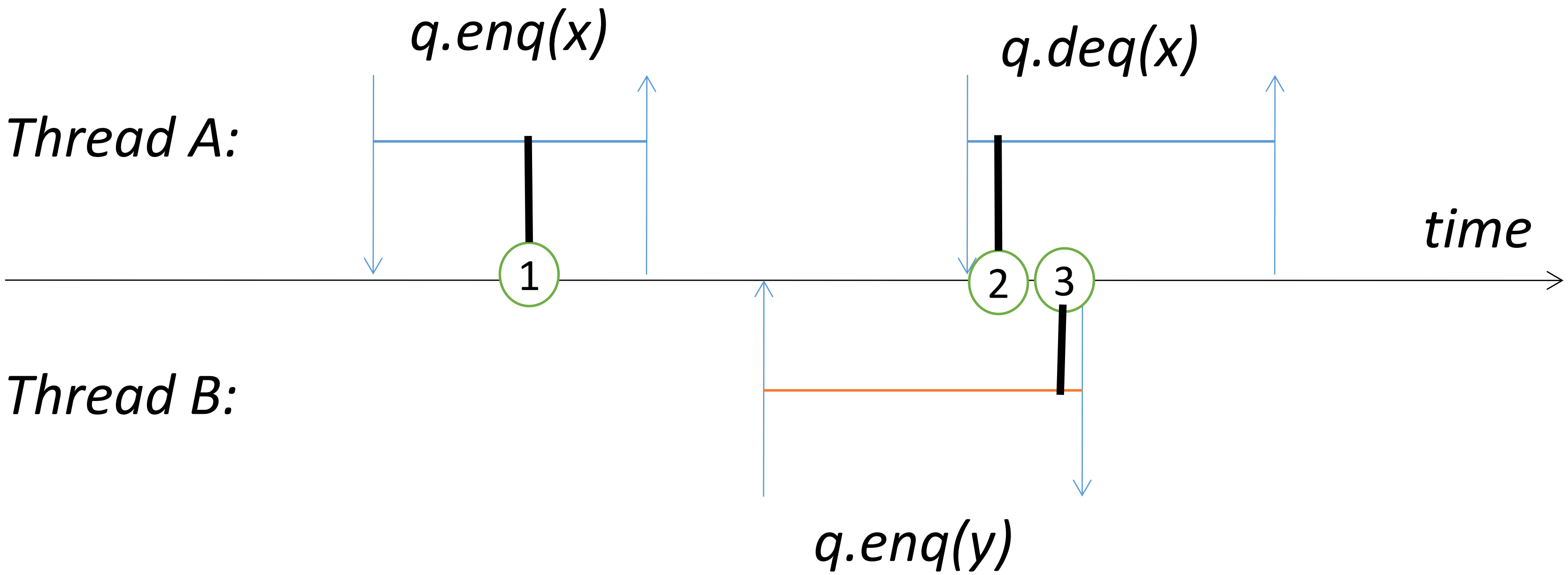
Total Order:

1. A: q.enc(x)
2. B: q.enq(y)
3. A: q.deq(x)

- Linearizability respects real time order

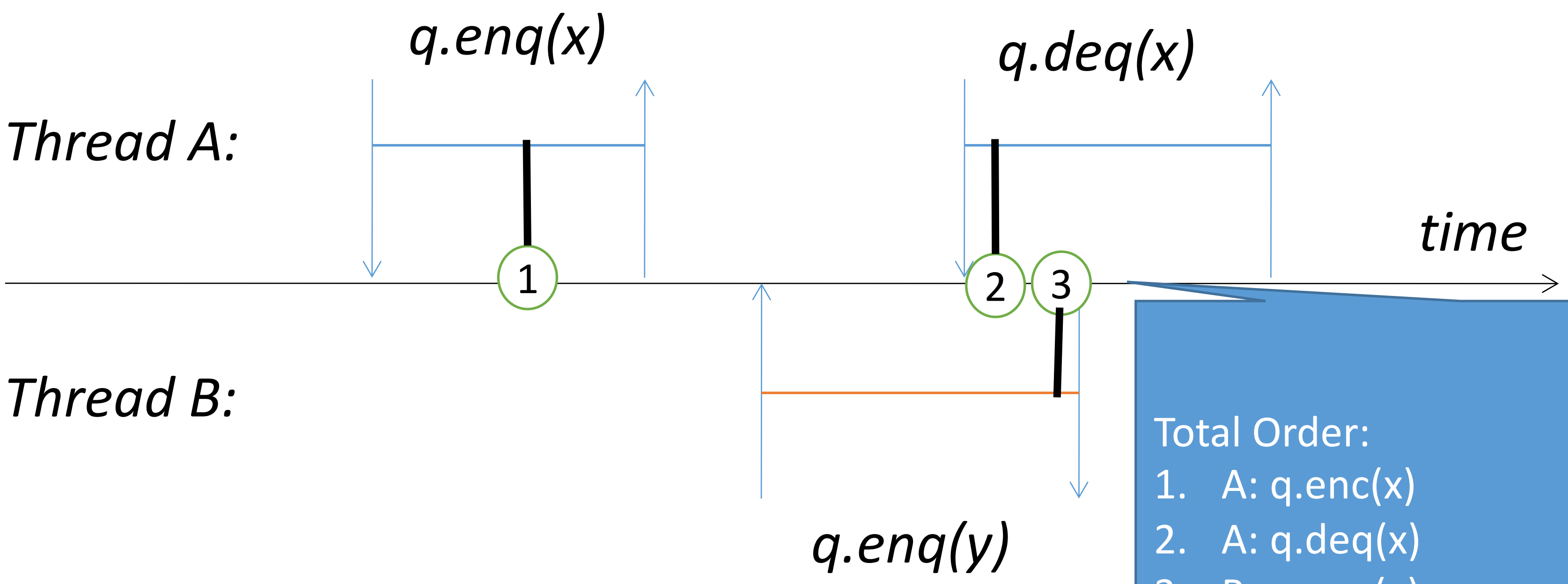
Linearizability, another interleaving

q is a FIFO queue



Linearizability, another interleaving

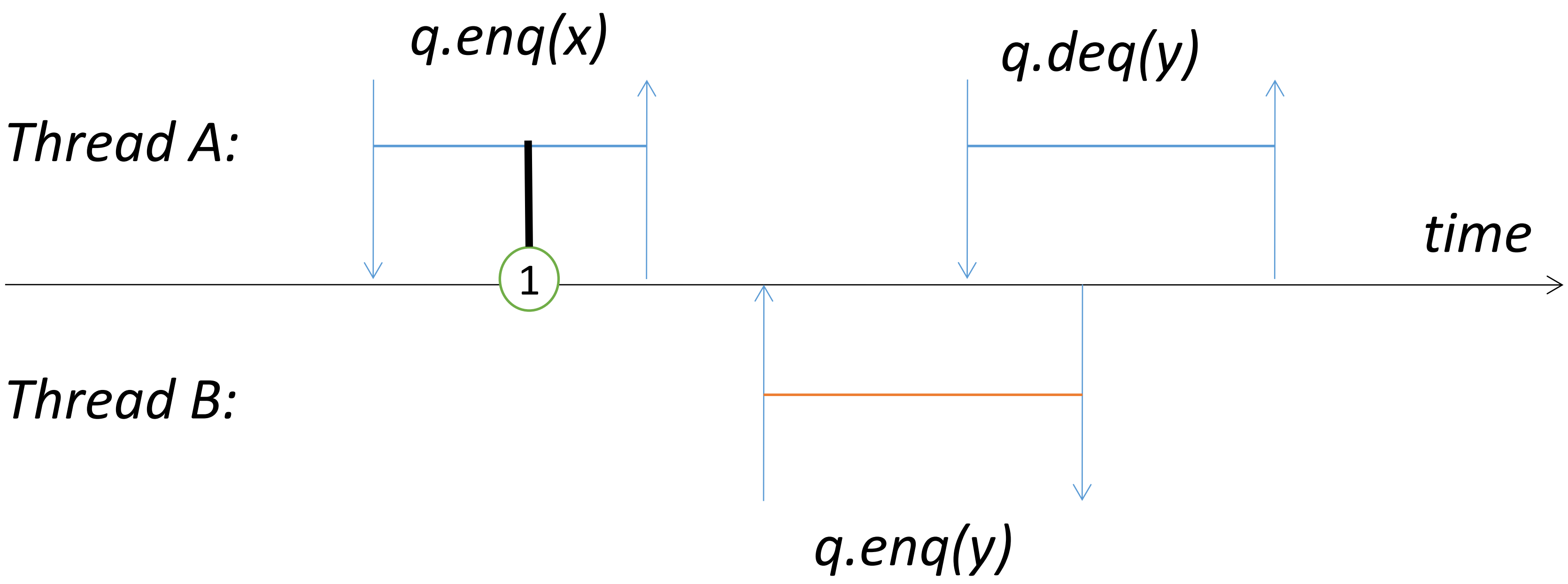
q is a FIFO queue



Total Order:
1. A: q.enc(x)
2. A: q.deq(x)
3. B: q.enq(y)

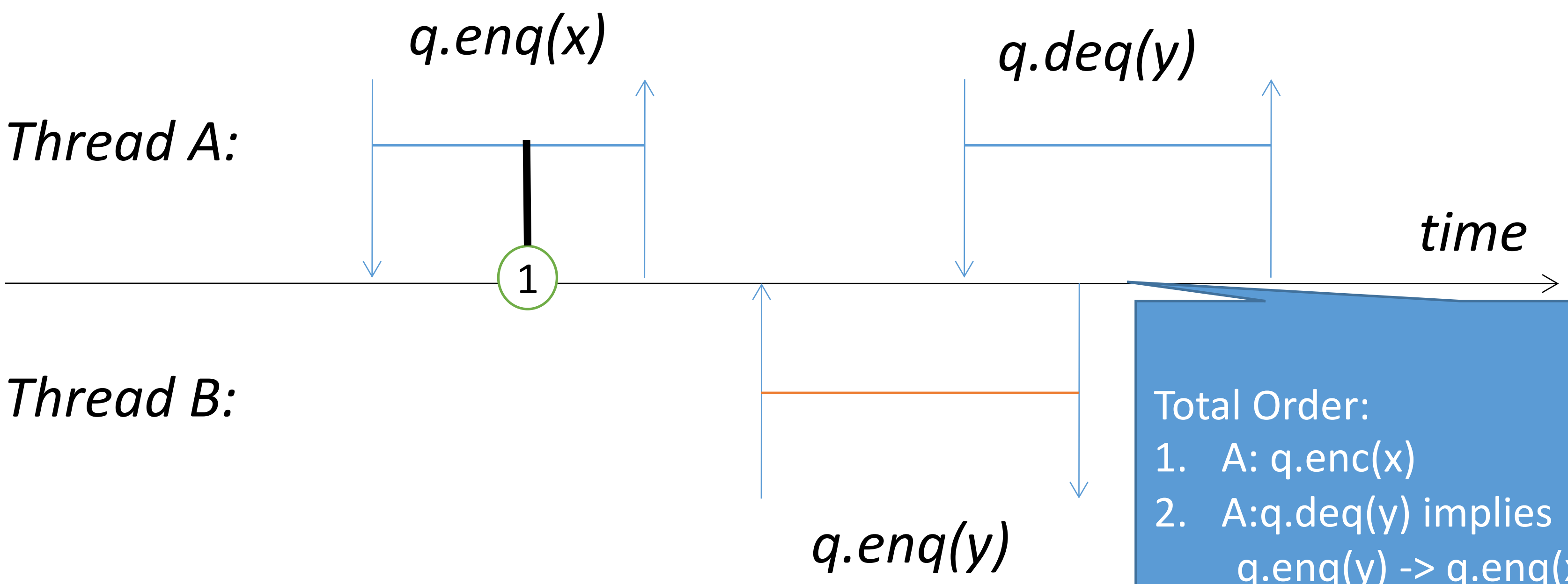
Not linearizable

q is a FIFO queue



Not linearizable

q is a FIFO queue



Total Order:

1. A: q.enc(x)
2. A:q.deq(y) implies
   q.enq(y) -> q.enq(x)

3. ????????

Recurring technique

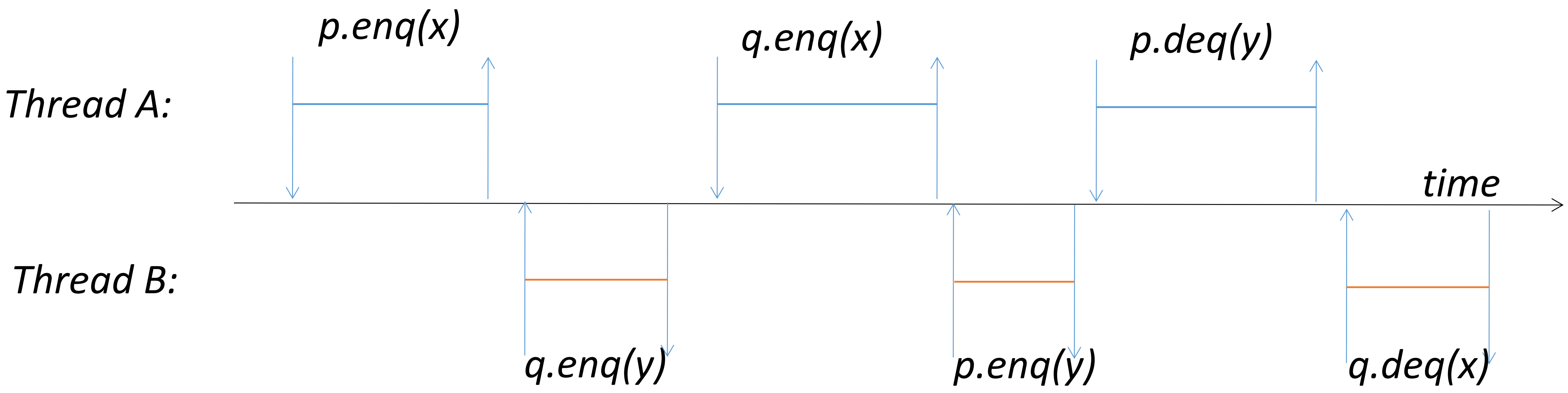
- For updates:
  - Perform an essential step of an operation by a single atomic instruction
  - E.g. CAS to insert an item into a list
  - This forms a “linearization point”
- For reads:
  - Identify a point during the operation’s execution when the result is valid
  - Not always a specific instruction

Linearizability vs. Sequential consistency

- So far, sequential consistency is weaker
    - SC allows more interleavings than linearizability
    - Higher performance, so why not always use it?

Sequential consistency not composable

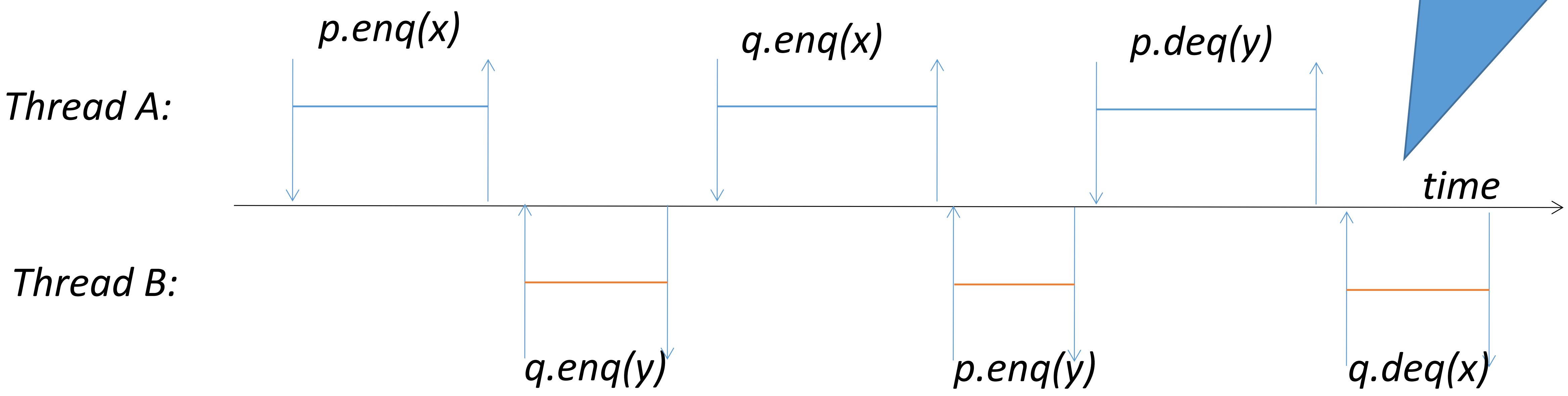
p&q are FIFO queues



Sequential consistency not comp

Total Order:
1. p is FIFO and A dequeues y from p, so y enqueued before x: ⟨p.enq(y) B⟩ → ⟨p.enq(x) A⟩
2. ⟨q.enq(x) A⟩ → ⟨q.enq(y) B⟩
3. Program order
- Cycle!

p&q are FIFO queues



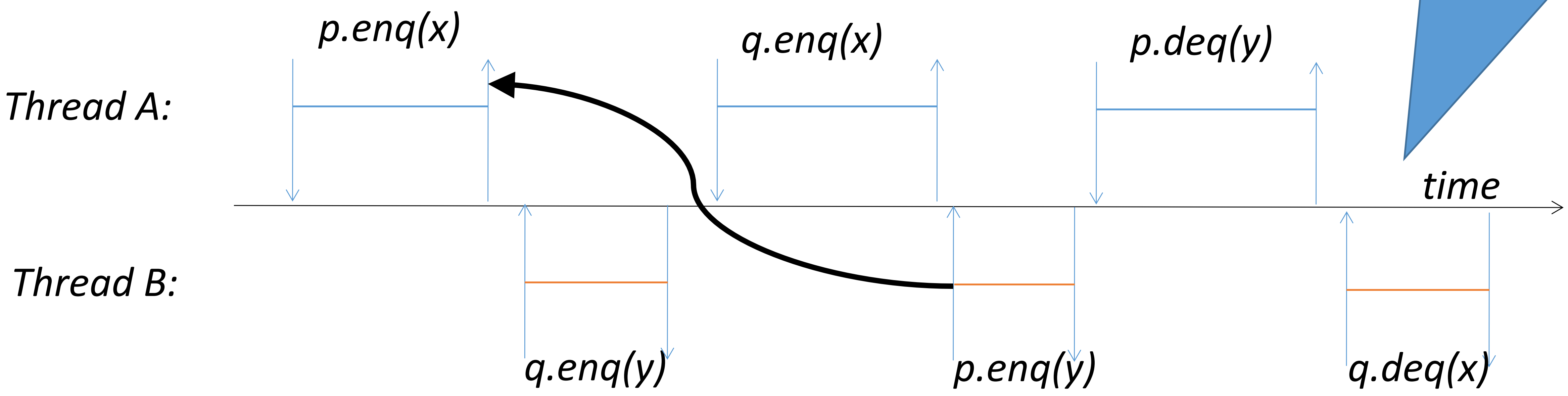
Sequential consistency not comp

Total Order:

1. p is FIFO and A dequeues y from p, so y enqueued before x: ⟨p.enq(y) B⟩ → ⟨p.enq(x) A⟩
2. ⟨q.enq(x) A⟩ → ⟨q.enq(y) B⟩
3. Program order

- Cycle!

p&q are FIFO queues



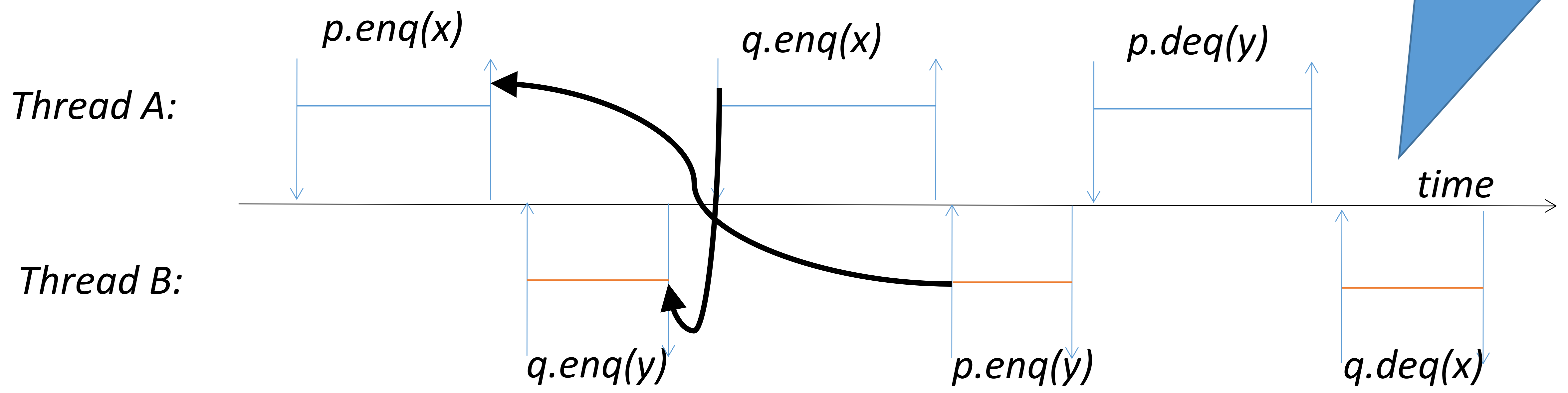
Sequential consistency not comp

Total Order:

1. p is FIFO and A dequeues y from p, so y enqueued before x: ⟨p.enq(y) B⟩ → ⟨p.enq(x) A⟩
2. ⟨q.enq(x) A⟩ → ⟨q.enq(y) B⟩
3. Program order

- Cycle!

p&q are FIFO queues



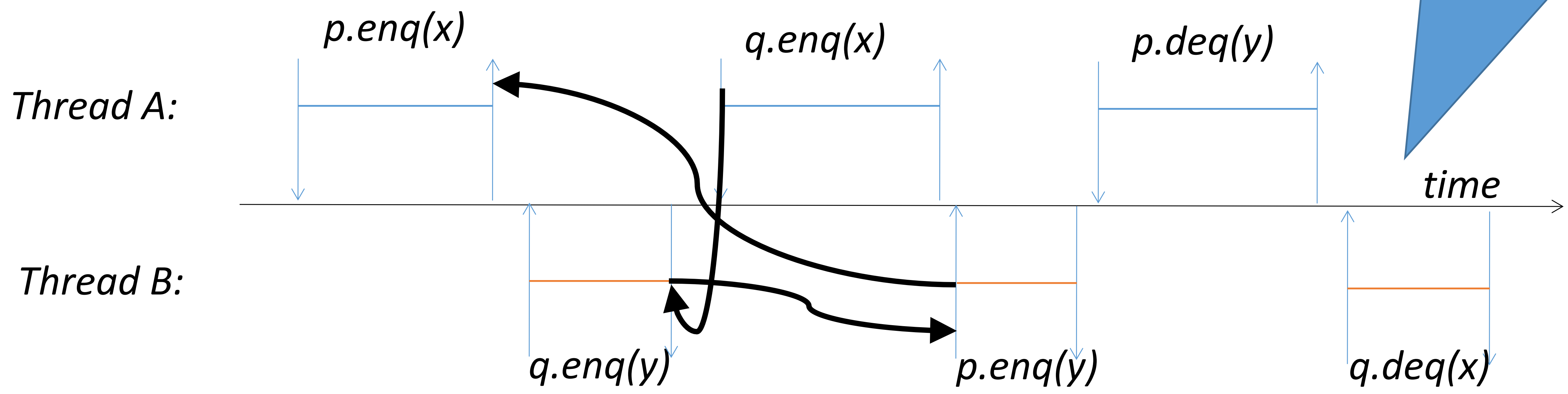
Sequential consistency not comp

p&q are FIFO queues

Total Order:

1. p is FIFO and A dequeues y from p, so y enqueued before x: ⟨p.enq(y) B⟩ → ⟨p.enq(x) A⟩
2. ⟨q.enq(x) A⟩ → ⟨q.enq(y) B⟩
3. Program order

- Cycle!



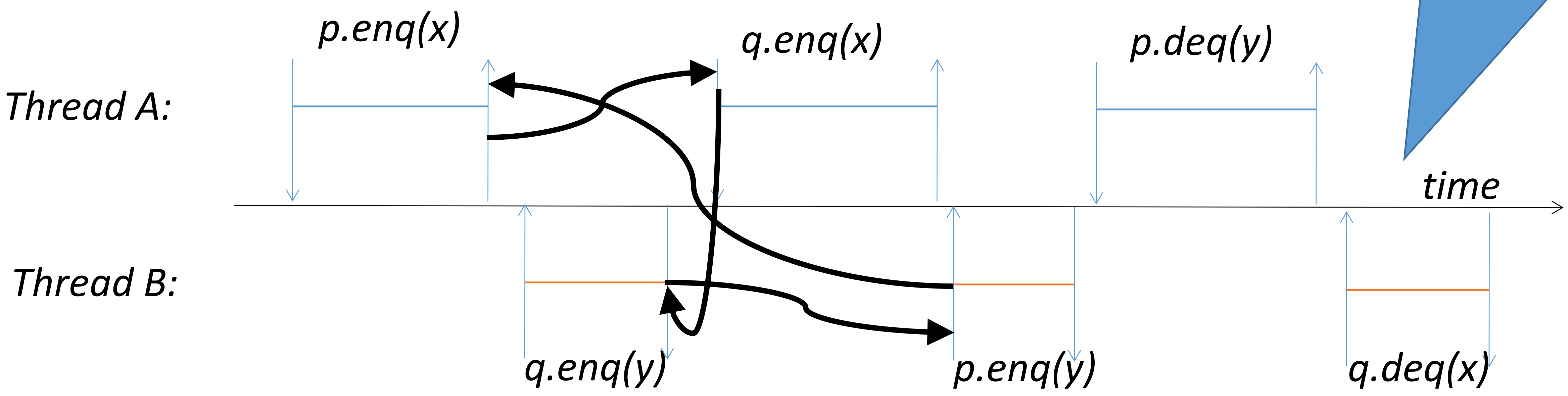
Sequential consistency not comp

Total Order:

1. p is FIFO and A dequeues y from p, so y enqueued before x: ⟨p.enq(y) B⟩ → ⟨p.enq(x) A⟩
2. ⟨q.enq(x) A⟩ → ⟨q.enq(y) B⟩
3. Program order

- Cycle!

p&q are FIFO queues



Sequential consistency

- SC is not composable.
  - A program that uses multiple SC objects is not necessarily SC
- So what is it good for?
  - When there is only 1 resource
  - E.g., DRAM
  - E.g., Fault-tolerant, distributed log
- Nothing to compose
- Violation of real-time order does not cause problems
  - Often because it is not “visible”

Defining concurrent queue implementations:

- Need a way to specify a concurrent queue object.
- Need a way to prove algorithms implement the specification.
- Concurrent specification imposes two new properties:
  - safety
  - liveness

Sequential vs. Concurrent

Sequential	Concurrent
Methods 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.