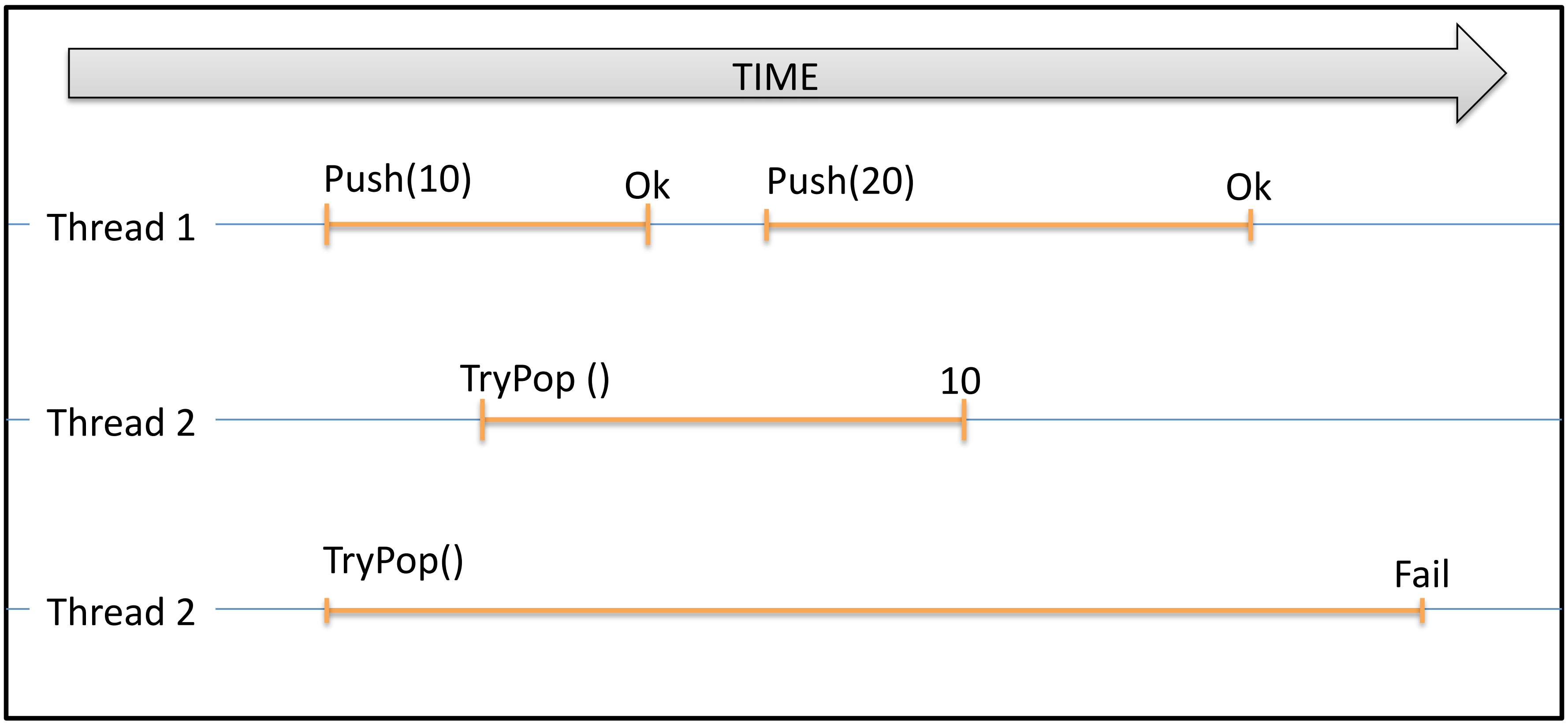
Review: Linearizability

Definition

- Given some component C (say, a class)
- And some operations O1, O2, .. (say, methods)
- *An operation is linearizable if it always appears to take effect at a single instant of time (called the commit point) which happens sometime after the operation is called and before it returns.*
- Linearizable operations are sometimes called *atomic,* but that term is overused.

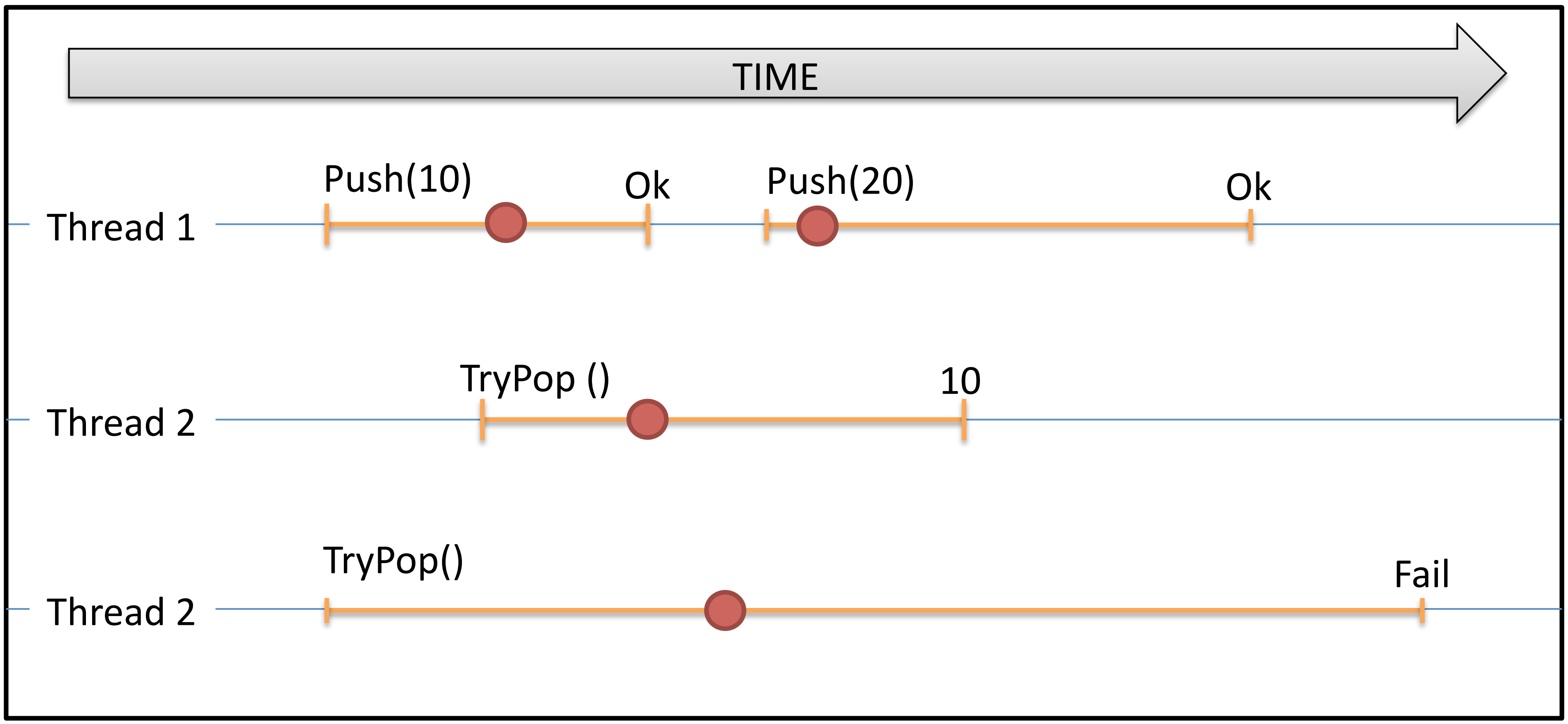
Example 1: Stack

- The following history is linearizable.



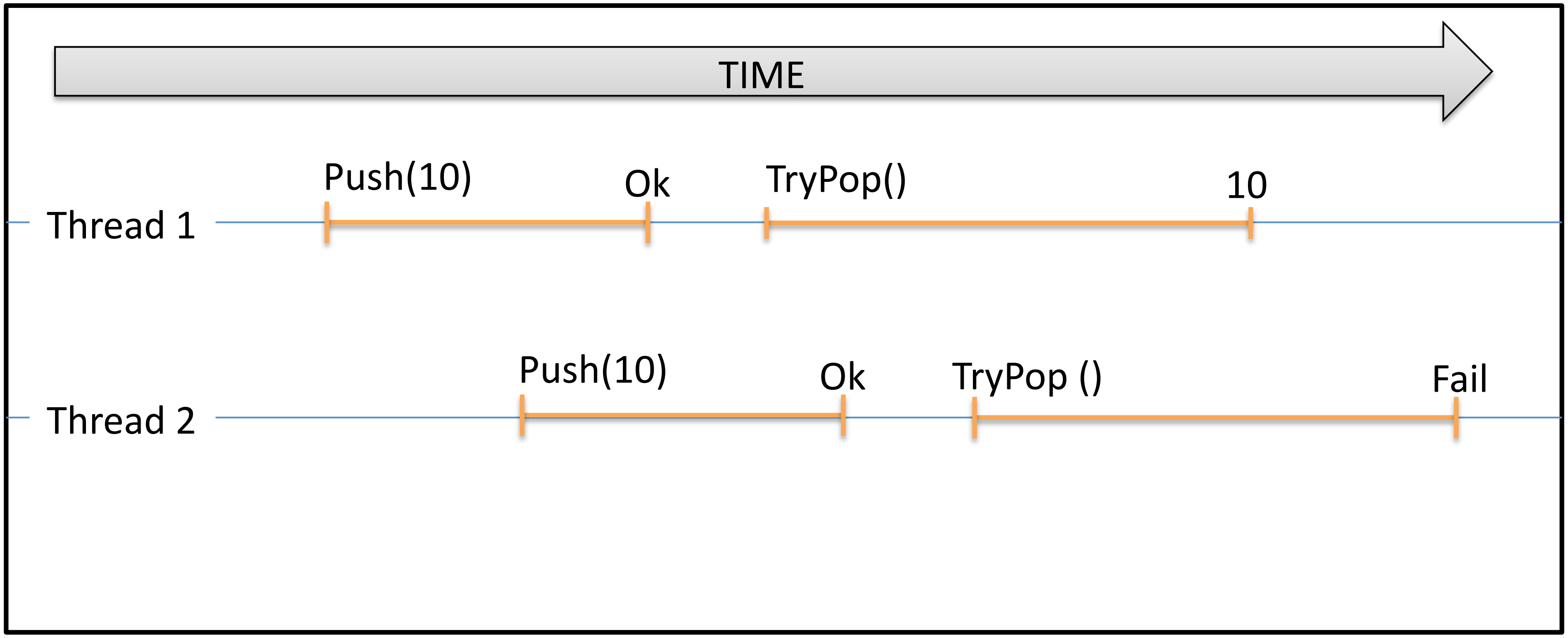
Example 1: Stack

- The following history is linearizable.



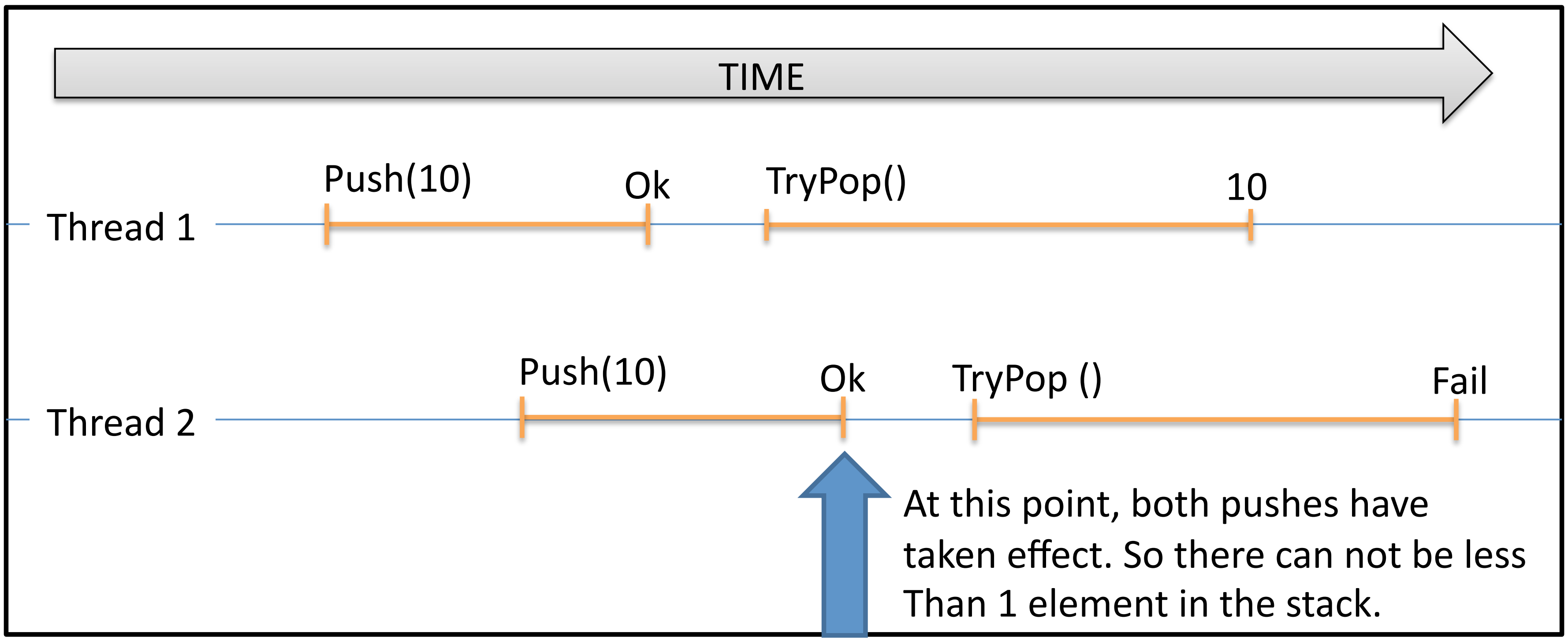
Example 2: Stack

- The following history is not linearizable.



Example 2: Stack

- The following history is not linearizable.



Quick Question

- Q: What is the most frequently used linearizable data type?

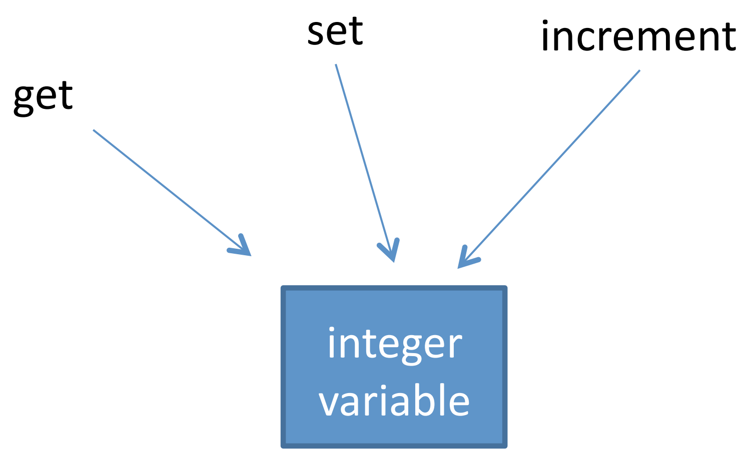
Quick Question

- Q: What is the most frequently used linearizable data type?
- A: an atomic register (historic name)

Example:

C = integer

O = { int get(),
void set(int val),
int increment() }



Plain fields and variables are not linearizable by default!

Atomic Registers in C#

- Use volatile declaration, e.g.

```
volatile int x;
```

  - Lets compiler know that you would like to read & write this field atomically. Important to avoid memory model issues.
  - Does not work with longs, structs
- Use “Interlocked” operations if you need an atomic modification
  - Interlocked.Increment, Interlocked.Decrement, Interlocked.Add
  - Interlocked.CompareExchange, Interlocked.Exchange
  - Interlocked.Read (for reading 64-bit longs)

Example: Volatile/Interlockeds Can Replace Locks

```
class MyCounter()
{
   Object mylock =  new Object();
   int balance;
   public void Deposit(int what)
    {
      lock(mylock)
           balance = balance + what;
    }
    public int GetBalance()
    {
       lock(mylock)
           return balance;
    }
    public void SetBalance(int val)
    {
        lock(mylock)
           balance = val;
    }
```

```
class MyCounter()
{
   volatile int balance;

   public void Deposit(int what)
   {
      Interlocked.Add(ref balance, what)
   }
    public int GetBalance()
    {
       return balance; /* volatile read */
    }
    public int GetBalance(int val)
    {
       balance = val; /* volatile write */
    }


}
```

The Composition Problem

- Atomic Registers & Linearizable Objects are great if you do 1 thing at a time.
- What if you need to do more than one thing at a time?

Stack Example

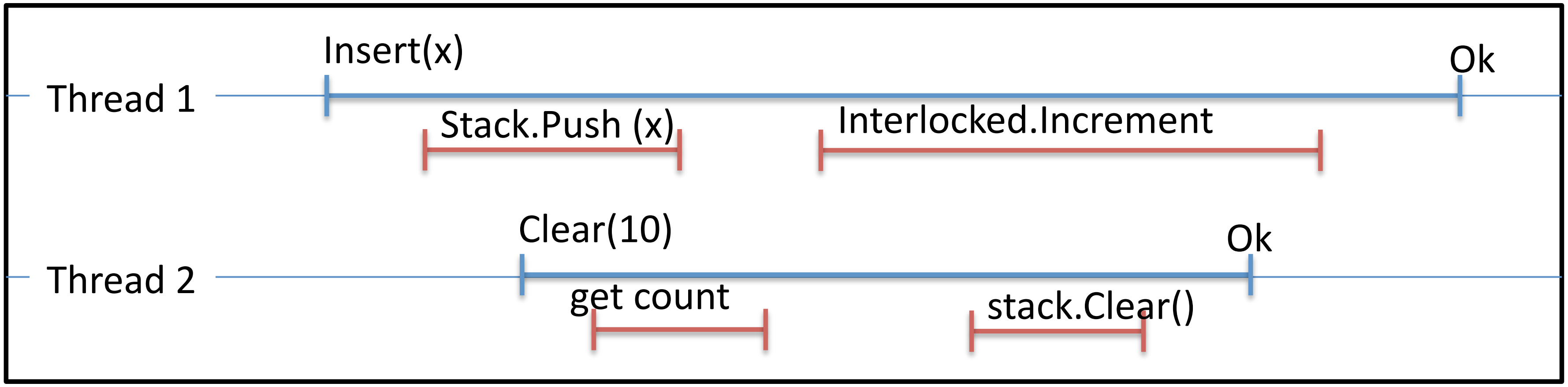
```
class SpecialStack
{
  LinearizableStack<Elt> stack;
  volatile int count;  // counts number of important elts in stack
}
```

Linearizable?

```
void Insert(x)
{
  stack.Push(x);
  if(x.Important)
    Interlocked.Increment(ref count);
}
```

```
bool Clear(x)
{
  if (count == 0)
  {
     stack.Clear();
     return true;
  }
  else
  {
     return false;
  }
}
```

Not linearizable.



Final state: stack empty, count=1

```
void Insert(x)
{
  stack.Push(x);
  if(x.Important)
    Interlocked.Increment(ref count);
}
```

```
bool Clear(x)
{
  if (count == 0)
  {
     stack.Clear();
     return true;
  }
  else
  {
     return false;
  }
}
```

Why so complicated? Just use a lock. Linearizability Restored.

```
class SpecialStack
{
  Stack<Elt> stack;
  int count;
}
```

```
void Insert(x)
{
   lock(this)
   {
     if(x.Important)
       count++;
     stack.Push(x);
   }
}
```

```
bool Clear(x)
{
  lock(this)
  {
    if (count == 0)
    {
       stack.Clear();
       return true;
    }
    else
    {
       return false;
    }
  }
}
```

Transactions & Concurrency Control

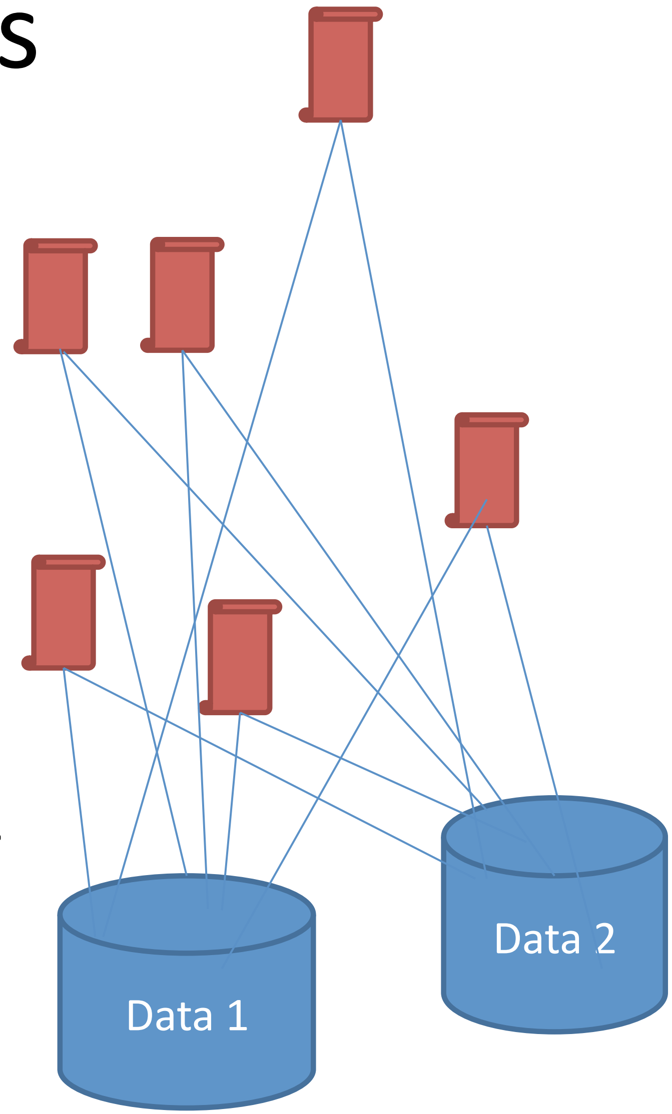
Unit 8.c

Acknowledgments

- Authored by
  - Sebastian Burckhardt, MSR Redmond

Transactions

- Clients vs. Data
  - Clients are concurrent (e.g. threads, processes, computers)
  - Data may be spread out (e.g. across processes, files, servers)
- Clients perform transactions
  - bounded sequence of operations `READ(location), WRITE(location, value)`
  - May include data-dependent branching or looping
  - May have real-world significance, e.g. represent a purchase or a reservation
  - What could possibly go wrong?



Example 1: Bank Accounts

Balance Inquiry

```
BEGIN_TRANSACTION
int x = READ(account1);
int y = READ(account2);
Print("total=", x+y);
COMMIT
```

Transfer 100 from account1 to account2

```
BEGIN_TRANSACTION
int x = READ(account1);
if (x >= 100)
{
    WRITE(account1, x-100);
    int y = READ(account2);
    WRITE(account2, y+100);
}
COMMIT
```

- If interleaved, may present incorrect total balance.

Example 2: Bank Accounts

Transfer 100 from account1 to account2

```
BEGIN_TRANSACTION
int x = READ(account1);
if (x >= 100)
{
    WRITE(account1, x-100);
    int y = READ(account2);
    WRITE(account2, y+100);
}
COMMIT
```

Transfer 100 from account1 to account2

```
BEGIN_TRANSACTION
int x = READ(account1);
if (x >= 100)
{
    WRITE(account1, x-100);
    int y = READ(account2);
    WRITE(account2, y+100);
}
COMMIT
```

- If interleaved, may lose or create money.

Atomicity Consistency Isolation Durability

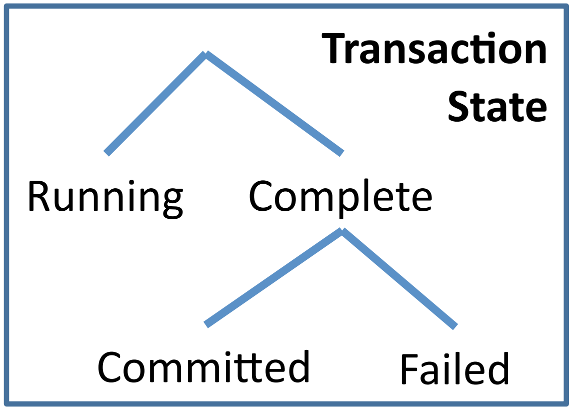
- ACID properties represent some common expectations on behavior of a transaction processing system.

- Databases implement basic ACID
- ACID is not a completely precise definition, but good to know for reference.

Transaction States

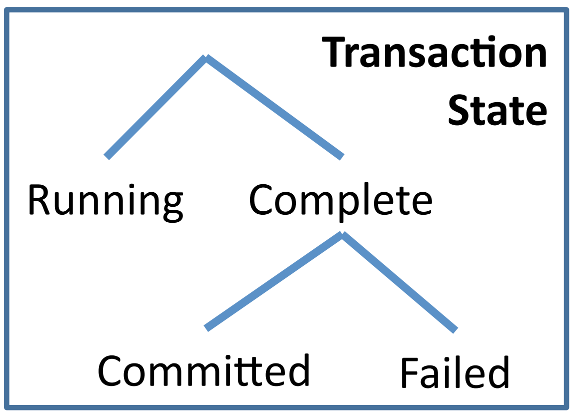
- Client program starts and ends transactions.

- Start transaction
  - Begins in state Running
  - Can read and modify data
- End transaction
  - Move to state Complete
  - System determines whether transaction commits or fails



Atomicity: All-or-nothing

- All changes by a committed transaction take effect
- No changes by a failed transaction take effect



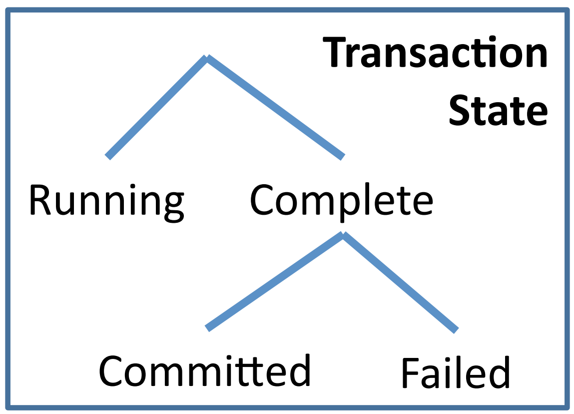
- (Note: the atomicity property refers to *Complete* transactions only, not *Running* transactions)

Consistency

- If a transaction starts in a consistent state, then it ends in a consistent state
- How do we define “consistent” ?
  - (A) “satisfies specific consistency properties”
    - such as declared by a database schema
    - such as general sanity conditions, e.g. no dangling pointers
  - (B) “satisfies design invariants required for correct program function”
    - those are not usually documented or even known
- Databases use definition (A)
  - Database must abort transactions that violate consistency

Isolation

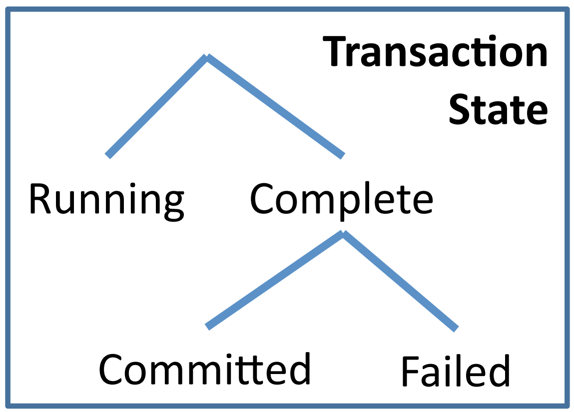
- A transaction may not observe changes made by another running transaction.



- That is, changes by a transaction A are not visible to a transaction B before A commits.

Durability

- Once committed, the effects of a transaction are permanent even if system failures occur.



- For some definition of 'system failure'.
  - For example, power outage.
- Database systems use disk-logs to guarantee this.

How strong is ACID?

- Not as strong as you may think.
- A+C+I does not add up to linearizability.

- Linearizability Definition:
  Successful transactions appear to execute without interruption, at a single instant of time (called the commit point) which happens sometime after the transaction starts and before it ends.

- Why is A+C+I not enough?

Non-Repeatable Read

`BEGIN_TRANSACTION`	
`int r1 = READ(x);`	
	`BEGIN_TRANSACTION`
	`WRITE(x,10)`
	`COMMIT`
`int r2 = READ(x);`	
`COMMIT`	

- Second read of x may return different value than first.
- This execution is not linearizable, but satisfies ACID !
  - Not equivalent to any sequential execution of the committed transactions
  - But Isolation is satisfied: reads see effects of committed transactions only
- Problem: the definition of I in ACID is too weak… we should consider alternatives.

Isolation Levels

Some Isolation Levels offered by commercial DB systems:

- READ_UNCOMMITTED (no isolation)
  - Can see changes of other running transactions
- READ_COMMITTED (weak isolation)
  - Can only see changes of committed transactions
- SNAPSHOT (strong isolation)
  - Work on isolated copy, then check for write conflicts at end
- SERIALIZABLE (more than isolation)
  - Pretty much the same as linearizability

    (technically, serializable is slightly weaker as commit points may be outside the transaction range)

TRANSACTIONS AND TRANSACTIONAL MEMORY

Using Transactions

- Consider more specific situation
  - Single multi-processor machine
  - Many threads operating on shared data
  - Threads want to perform linearizable transactions
- Can we use a DB to do the work for us?
  - Yes, if it performs well enough and isn't too expensive.
- But how could we do it from scratch?

Software Transactional Memory (STM)

- Software Transactions are the “universal linearizable datatype” – they assume no particular data structure, nor a particular access pattern.
- STMs not actually common in practice.
  - Despite loads of research
- But: Understanding STM is an excellent exercise for building linearizable components.

Outline

- Let's build a simple but fully functional STM
  - (full code on codeplex)
- Several steps:
  - Define a transaction API
  - Build a wrong implementation
  - Build a lock-based pessimistic implementation (2-phase locking)
  - Build a simple optimistic implementation (speculate on absence of conflicts)

Simple Transaction API

```
public class TransactionProcessor
{
    // start a new transaction
    public Transaction StartTransaction() { … }
}

public class Transaction
{
    // read from the given location
    int ReadLocation(Location l) { … }

    // write to the given location
    void WriteLocation(Location l, int value) { … }

    // try to commit transaction
    void Commit() { … }

    // abort this transaction
    void Abort() { … }
}

// thrown by { ReadLocation, WriteLocation, Commit }
public class TransactionFailedException : Exception {  … }
```

How to use API

Transfer 100 from acc1 to acc2

```
BEGIN_TRANSACTION
int x = READ(acc1);
if (x >= 100)
{
    WRITE(acc1, x-100);
    int y = READ(acc2);
    WRITE(acc2, y+100);
}
COMMIT
```

```
Transaction t = p.StartTransaction();
try
{
   int x = t.ReadLocation(acc1);
   if (x >= 100)
   {
      t.WriteLocation(acc1, x - 100);
      int y = t.ReadLocation(acc2);
      t.WriteLocation(acc2, y + 100);
   }
   t.Commit();
}
catch (TransactionFailedException)
{
   …
}
```

Conflicts & Concurrency

- Classify conflicts
  - Read-Write conflict
    Transaction A writes to the same location that Transaction B reads from.
  - Write-Write conflict
    Transaction A and B both write to the same location.
- Transactions without conflicts can execute concurrently (at least in principle)
- Transactions with conflicts need more caution
- *Don't know in advance if transactions conflict!*
  - Can use locks to order conflicts.

1st Implementation: "Pessimistic" Concurrency Control

- Protect locations using locks
  - One lock per location, or
  - One lock for all locations, or
  - One lock for a group of locations
- Ensure you hold lock while reading or writing a location.
- Is that enough?

Naïve implementation (BROKEN)

```
class Transaction
{
    // read from the given location
    public int ReadLocation(Location l)
    {
        lock (l)
        {
            return l.value;
        }
    }
    // write to the given location
    public void WriteLocation(Location l, int value)
    {
        lock (l)
        {
            l.value = value;
        }
    }
    // try to commit transaction
    public void Commit()
    {
    }
}
```

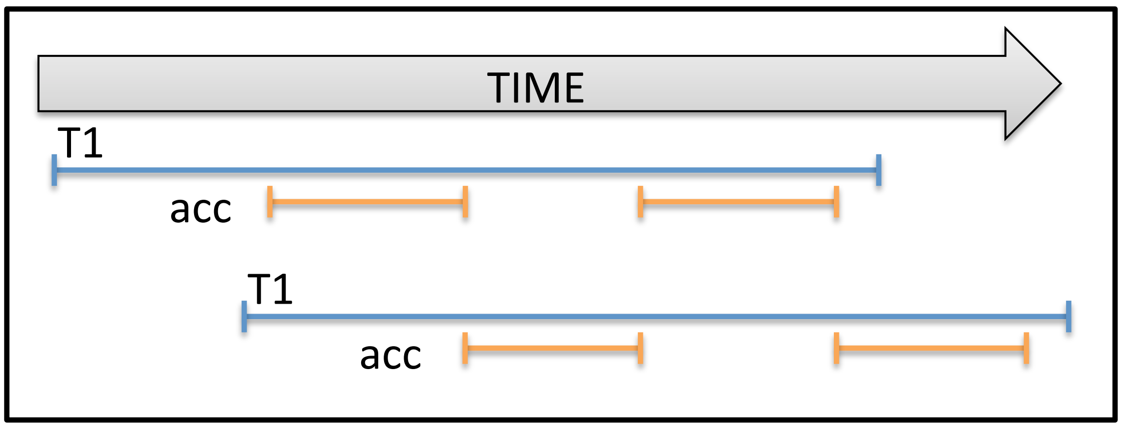
Visualization of Broken Implementation

Transaction 1

```
BEGIN_TRANSACTION
int x = READ(acc);
WRITE(acc, x+100);
COMMIT
```

Transaction 2

```
BEGIN_TRANSACTION
int x = READ(acc);
WRITE(acc, x+100);
COMMIT
```



Blue Segments:
Transactions (begin/end)

Orange Segments:
Locks (acquire/release)

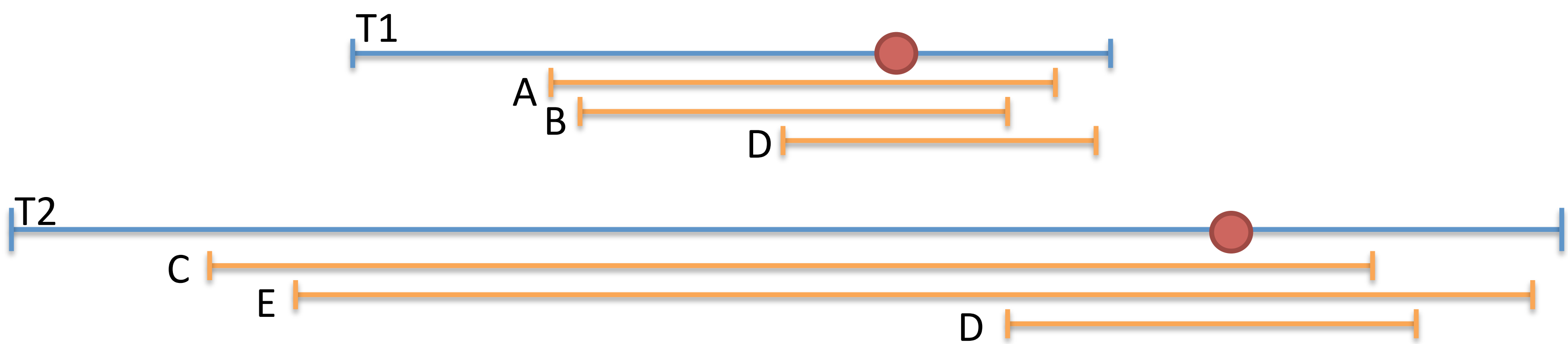
- Need to hold locks long enough to guarantee atomicity!

2-Phase Locking

- All locations are protected by some lock.
- Transactions can access locations only while holding their lock.
- Transactions must follow 2 phases
  - Expanding phase: May acquire new locks but not release any held locks
  - Shrinking phase: May release held locks but not acquire any new locks
- Following this protocol guarantees linearizability!

[Bernstein et al. 1987].

2-Phase Locking Illustration



Time: left to right
Blue Segments: Transactions (begin/end)
Orange Segments: Locks (acquire/release)
Red circles: Commit Points

- Commits while holding all locks of all accessed locations
- Therefore, all read & written values consistent with commit order

Simple 2PL-implementation (1/2)

```
class Transaction
   {
       // store current set of held locks
       HashSet<Location> locks_held = new HashSet<Location>();

       // read from the given location
       public int ReadLocation(Location l)
       {
           if (!locks_held.Contains(l))
           {
               System.Threading.Monitor.Enter(l);
               locks_held.Add(l);
           }
           return l.value;
       }
```

Simple 2PL-implementation (2/2)

```
// write to the given location
public void WriteLocation(Location l, int value)
{
    if (!locks_held.Contains(l))
    {
        System.Threading.Monitor.Enter(l);
        locks_held.Add(l);
    }
    l.value = value;
}

// try to commit transaction
public void Commit()
{
    // shrinking phase… release all the locks
    foreach (Location l in locks_held)
        System.Threading.Monitor.Exit(l);
}
```

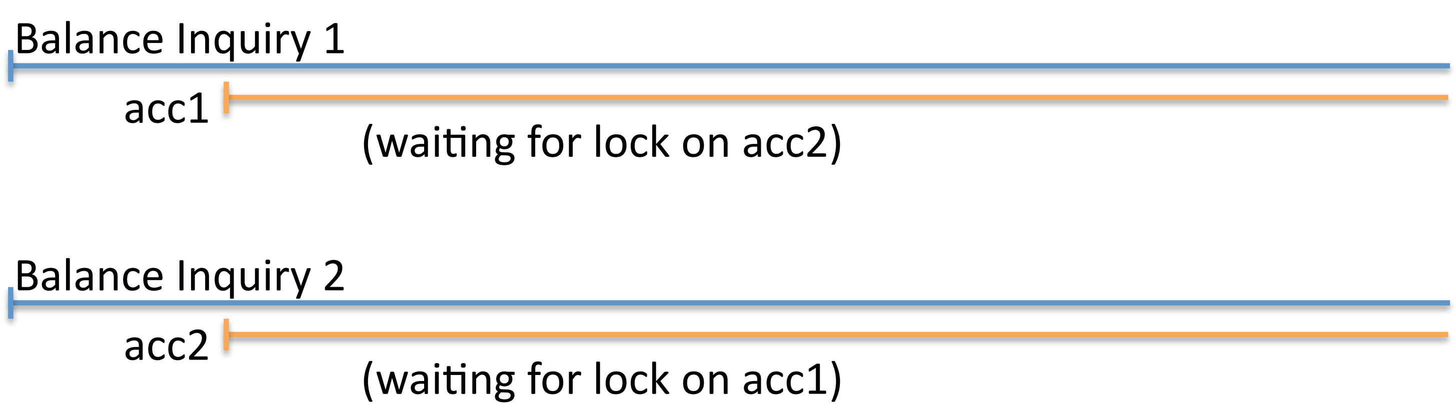
Simple 2PL-implementation BUSTED: Deadlock Example

Balance Inquiry 1

```
BEGIN_TRANSACTION
int x = READ(account1);
int y = READ(account2);
Print(“total=“, x+y);
COMMIT
```

Balance Inquiry 2

```
BEGIN_TRANSACTION
int y = READ(account2);
int x = READ(account1);
Print(“total=“, x+y);
COMMIT
```



Simple 2PL-implementation FIXED (1/2): Time Out and Abort

```
const int LOCK_TIMEOUT_MILLISECONDS = …;

public void TryAcquire(Location l)
{
    if (System.Threading.Monitor.TryEnter(l, LOCK_TIMEOUT_MILLISECONDS))
    {
        locks_held.Add(l);
    }
    else
    {
        Abort();
        throw new TransactionFailedException("lock timed out");
    }
}
public void Abort()
{
    // first, restore old values
    foreach (KeyValuePair<Location, int> kvp in savedvalues)
        kvp.Key.value = kvp.Value;

    // then, release all the locks
    foreach (Location h in locks_held)
        System.Threading.Monitor.Exit(h);
}
```

Simple 2PL-implementation FIXED (2/2): Time Out and Abort

```
// store overwritten values in case we need to roll back
Dictionary<Location, int> savedvalues = new Dictionary<Location, int>();

// write to the given location
public void WriteLocation(Location l, int value)
{
    if (!locks_held.Contains(l))
        // we are not already holding a lock on this.. try to acquire it
        TryAcquire(l);

    // save old value if this is the first write by this transaction
    if (!savedvalues.ContainsKey(l))
        savedvalues[l] = l.value;

    l.value = value;
}
```

2PL implementation now works.

- Linearizable and simple.
- Some things are not so nice though:
  - Does not allow concurrent reads.
  - May keep locations locked for a pretty long time.
- Can we write an implementation with less locking and more concurrency?

Optimism vs. Pessimism

Suppose conflicts are rare.

- For many workloads, most writes go to locations that are not at the same time being read or written by another transaction.
- We can use *speculation*: Execute transaction optimistically (i.e. elide locking and hope there are no conflicts), keeping changes in a 'sandbox'
- If speculation fails, abort transaction and discard changes. Otherwise, make changes permanent.

Simple optimistic implementation (1/4)

```
class Transaction
{

       // temporary data for transaction. For each location accessed:
       // - stores first value read if first access was a read
       // - stores last value written
       SortedDictionary<Location, Entry> scratch = new
                                           SortedDictionary<Location,Entry>();


       class Entry
       {
           // has this transaction written to this location?
           // if yes, what was the last value written?
           public bool written;
           public int last_value_written;

           // was the first access by this transaction a read?
           // if yes, what value was read?
           public bool first_access_was_read;
           public int value_read;
       }
```

Simple optimistic implementation (2/4)

```
// read from the given location
public int ReadLocation(Location l)
{
    Entry s;

    // if this location is not in scratch, put it there.
    if (! scratch.TryGetValue(l, out s))
    {
        s = new Entry();
        s.first_access_was_read = true;
        s.value_read = l.value;
        scratch[l] = s;
    }

    return (s.written ? s.last_value_written : s.value_read);
}
```

Reads volatile field *without lock*

```
class Entry
{
    // has this transaction written to this location?
    // if yes, what was the last value written?
    public bool written;
    public int last_value_written;

    // was the first access by this transaction a read?
    // if yes, what value was read?
    public bool first_access_was_read;
    public int value_read;
}
```

Simple optimistic implementation (3/4)

```
// write to the given location
public void WriteLocation(Location l, int value)
{
    Entry s;

    // if this location is not in scratch, put it there.
    if (!scratch.TryGetValue(l, out s))
    {
        s = new Entry();
        s.first_access_was_read = false;
        scratch[l] = s;
    }

    s.last_value_written = value;
    s.written = true;
}
```

Writes value to temp storage only, *without lock*

```
class Entry
{
   // has this transaction written to this location?
   // if yes, what was the last value written?
   public bool written;
   public int last_value_written;

   // was the first access by this transaction a read?
   // if yes, what value was read?
   public bool first_access_was_read;
   public int value_read;
}
```

Simple optimistic implementation (4/4)

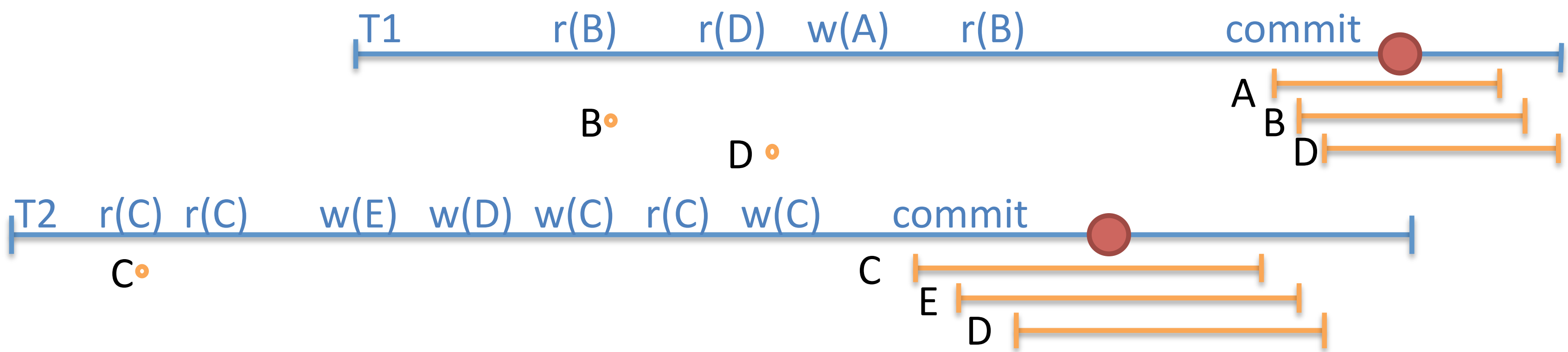
```
// try to commit transaction using 2-phase locking.
public void Commit()
{
    bool failed = false;

    // phase 1 (expanding) grab all locks, and validate reads
    foreach (KeyValuePair<Location, Entry> kvp in scratch)
    {
        // acquire lock (no deadlock since ordering is respected)
        System.Threading.Monitor.Enter(kvp.Key);

        // if this location was read, check if value would read the same right now
        if (kvp.Value.first_access_was_read && kvp.Value.value_read != kvp.Key.value)
            failed = true;
    }
    // phase 2 (shrinking) release all locks, and make writes permanent
    foreach (KeyValuePair<Location, Entry> kvp in scratch)
    {
        // if this transaction is successful, write back the last value written
        if (!failed && kvp.Value.written)
            kvp.Key.value = kvp.Value.last_value_written;

        // release lock
        System.Threading.Monitor.Exit(kvp.Key);
    }
    if (failed)
        throw new TransactionFailedException("optimism failure - read value changed");
}
```

Illustration



Blue Segments: Transactions (begin/end)
Orange Segments: Locks (acquire/release)
Orange Dots: Volatile Loads
Red circles: Commit Points

- First read access samples value
- Everything else is completely isolated until commit
- Commit replays all reads and makes all writes permanent

Other STM implementations

- Well-known optimistic implementations exist that are faster than the simple one we just looked at
- Example: TL2 algorithm by Dice, Shalev, Shavit
  - Does not hold locks for read locations during commit
  - Uses global version clock, and version number for each location to detect conflicts
  - Uses Bloom filter to check set membership efficiently (with nonzero one-sided error probability)
  - Does not order locks, but handles deadlock with time-out and abort ("sorting write-sets was not worth the effort").

Recap
Pessimistic vs. Optimistic

- Pessimistic Concurrency Control
  - Use locks to prevent conflicts
  - If deadlocked, roll back changes and abort
  - *Example*: 2-Phase Locking

- Optimistic Concurrency Control
  - Proceed *speculatively* (assume no conflicts), and keep all changes separate
  - At commit time, detect conflicts
  - If conflicts, roll back changes (if necessary) and abort
  - *Examples*: replay-reads-algorithm, TL2 algorithm (Dice, Shalev, Shavit)

Optimistic Concurrency Control is not a Panacea

- Doesn't work with permanent side effects
  - Can not always roll back (e.g. dispense cash on ATM)
- Conflicts aren't always rare
  - Some tasks always conflict
  - If conflicts are frequent, pessimistic performs better
- But don't despair: there is another solution that works in those cases: Concurrent Revisions