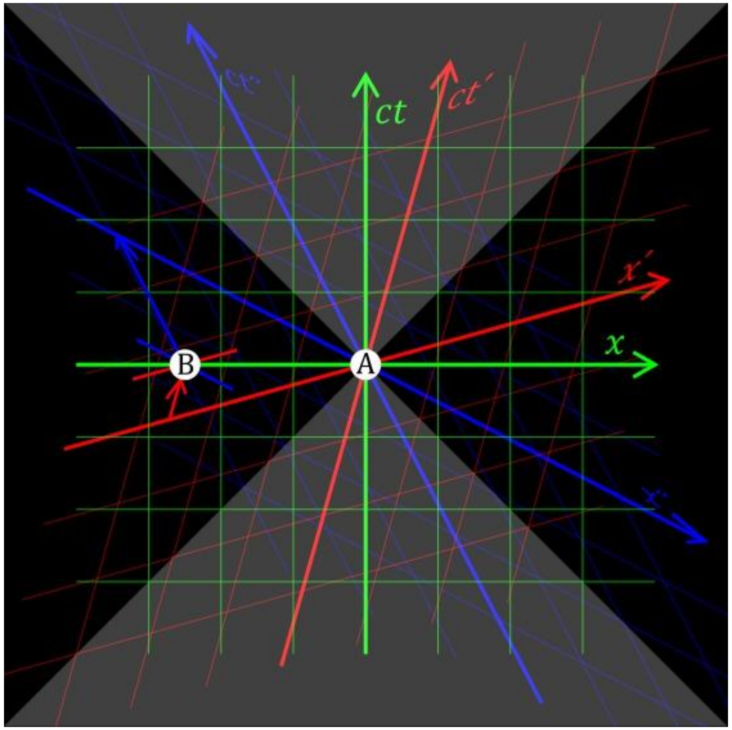
# ACID: The Wrong Way To Think About Concurrency

Emmett Witchel
The University of Texas At Austin

- Q: When is everything happening?




- A: Now

- A: Concurrently

# Concurrency is central to CS

- CS is at forefront of understanding concurrency
  - We operate near light speed
- Concurrent computer systems ubiquitous
  - Multiprocessors
  - Distributed systems
  - Data centers
- Great recent progress, but more to go

# What is a concurrent program?

# Concurrent counter increment

```
load c to reg
increment reg
store reg to c
```

```
load c to reg
increment reg
store reg to c
```

- Green and blue thread increment counter
  - Each thread on different processor
  - Threads share memory
- So $c_{final} == c_{init} + 2$

# Some interleavings are bad

```
load c to reg
increment reg
store reg to x
```

```
load c to reg
increment reg
store reg to c
```

Global order

```
load c to reg
load c to reg
increment reg
increment reg
store reg to c
store reg to c
```

- Some parallel executions are wrong
  - A bad interleaving causes $c_{final} == c_{init}+1$
- Critical region needs special handling

# Critical region

- Critical region – a code region requiring special properties to protect it from concurrent execution of other code

```
begin critical region
load c to reg
increment reg
store reg to x
end critical region
```

# Basis for critical r

- Computer system
  - State machine + Comm
- Communication can h
  - Direct (messages)
  - Indirect (memory)
- But some states shou
- Concurrency a sword

# My agenda

- Problem: concurrency, critical regions
- Solution
  - Transactions
  - Transaction processing system (TPS)
- Define the ACID properties
  - ACID != transactions
  - ACID is a single point, let's see the space
- Scheduling concurrency
  - Understand concurrency by eliminating it

# What are transactions?

- A transaction defines a critical region
  - Begin transaction
  - End transaction
- A transaction processing system (TPS) specifies proper concurrent execution
- Transactions and TPS as generic concurrency control
  - Possibly the “simplest” idea to specify concurrency

# Transactions aren’t a thing

- Less specific than an algorithm
- Less specific than a data structure
- Less specific than a design pattern

# Database example

- Transaction procedures
  - txbegin
  - txend
  - txabort
- Transactions read and write rows and tables

```
BEGIN TRAN T1;
UPDATE table1 ...;
SELECT * from table1;
COMMIT TRAN T1;
```

# Transactional memory example

- Transaction instructions
  - txbegin
  - txend
- Transactions read and write cache lines
  - Increment a counter

```
    lea c, %rax
retry:
    SPECULATE
    jnz retry
    lock mov (%rax), %rbx
    incr %rbx
    lock mov %rbx, (%rax)
    COMMIT
```

# The story so far

- Some code is vulnerable to other concurrently executing code
- Delimit critical region as a transaction
- Execute transaction in TPS
- WIN!
- …but what is the transaction processing system (TPS) supposed to do?
  - Traditional response: provide ACID properties
  - My response: schedule transactions

# ACID Properties

- **Consistency is Data structure invariants hold**
- Consistency: The transaction preserves the internal consistency of the database
- Isolation: The transaction executes as if it were running alone, with no other transactions
- Durability: The transaction's results will not be lost in a failure [B&N 2009]

# Database invariants

- Some can be maintained by system
  - E.g., referential integrity, roughly no dangling pointers
  - E.g., primary key values are unique
- Some externally enforced
  - E.g., Salary cannot decrease unless demotion
  - E.g., Number of widgets in DB equals physical widgets in warehouse

# Memory invariants

- Processor (ISA) invariants
  - E.g., 64-bit writes are indivisible
- Most externally enforced
  - E.g., List pointers correct
    - node->next->prev == node
  - E.g., Total items on list kept up to date with list membership

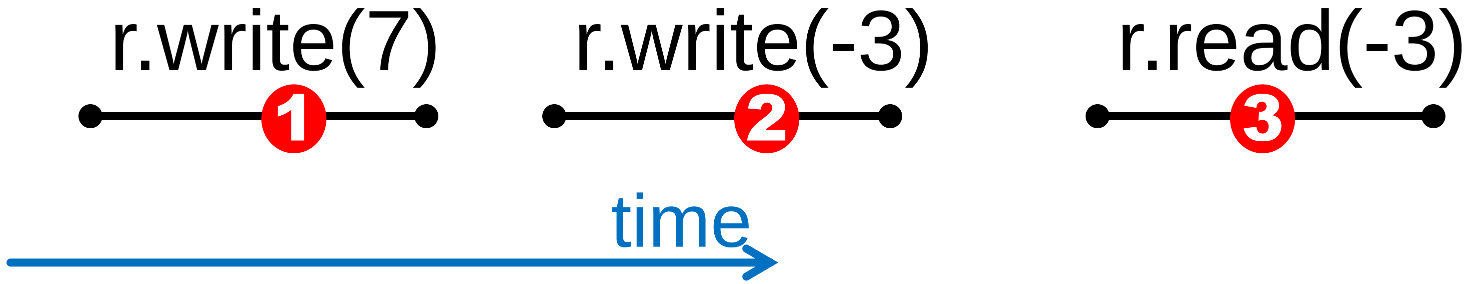
# Consistency not part of TPS

- A transaction system can't guarantee consistency!
  - A transaction can violate a data structure invariant
- …the transaction processing system does its part for the C in ACID only by guaranteeing AID. [B&N 2009]
- It's the application programmer's responsibility to ensure the transaction program preserves consistency. [B&N 2009]

# Isolation (from wikipedia)

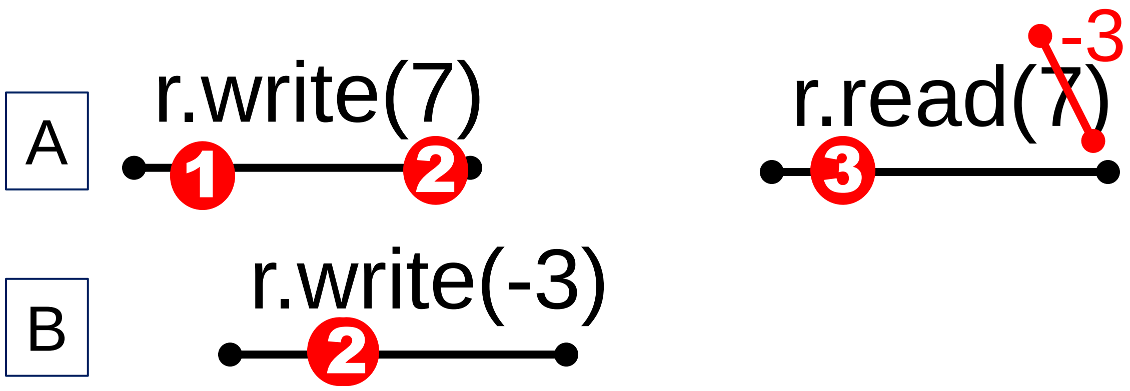
- Isolation refers to the requirement that no transaction should be able to interfere with another transaction. One way of achieving this is to ensure that no transactions that affect the same rows can run concurrently, since their sequence, and hence the outcome, might be unpredictable. This property of ACID is often partly relaxed due to the huge speed decrease this type of concurrency management entails.[citation needed]

# Schedule [H&S '08]




- A schedule consists of method invocations and responses (also called a history)
- A scheduler generates legal global orders
  - E.g., Methods should appear to happen in a one-at-a-time, sequential order

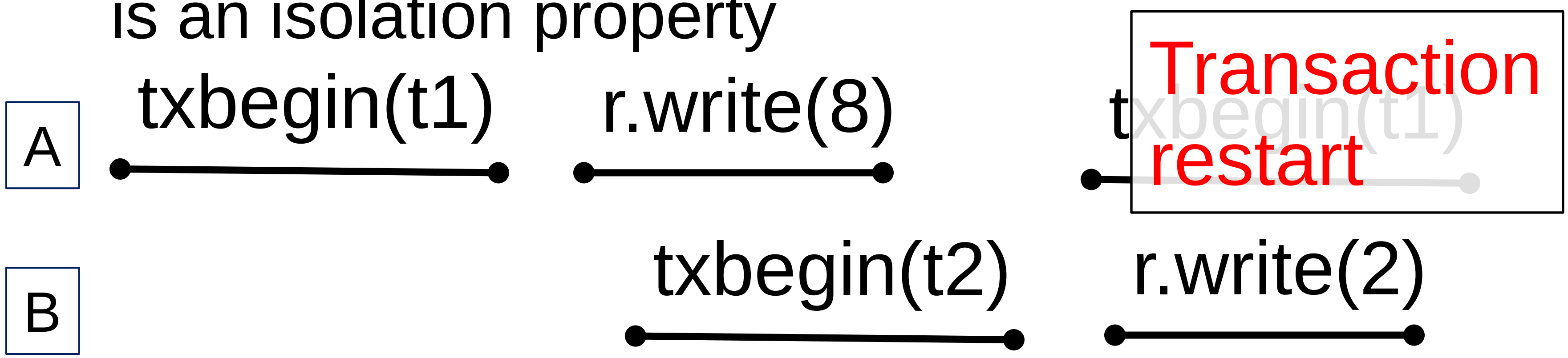
# Scheduling concurrency




- Many schedules are legal
  - r.read(-3) would also be "correct"
  - But reads return latest writes
- Scheduler defines safety and liveness
  - E.g., sequential consistency, serializability
  - E.g., r.read(-7) too weak for most TPSs

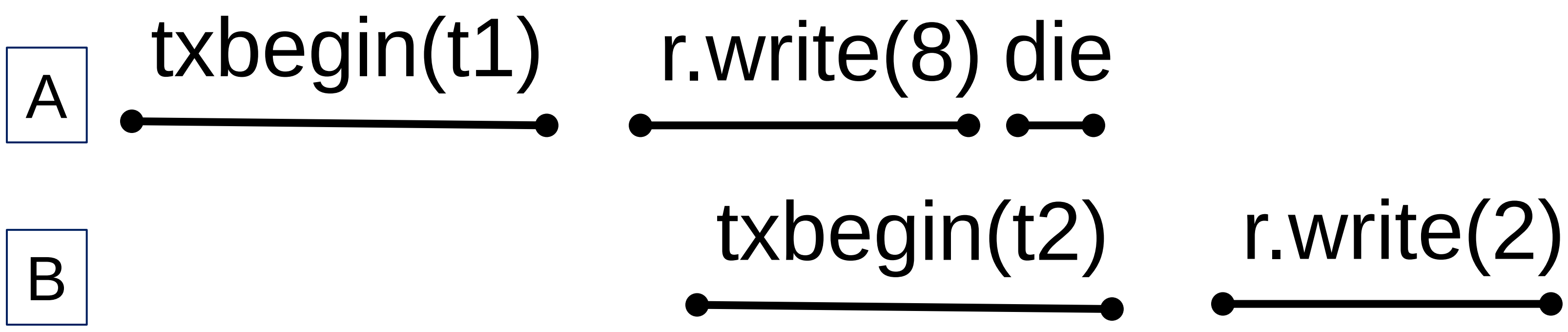
# Atomicity or Isolation?

- Two threads conflict…
  - Restart for atomicity - it must appear that either all of A's operations happened, or none.
  - Restart for isolation - not seeing partial results is an isolation property

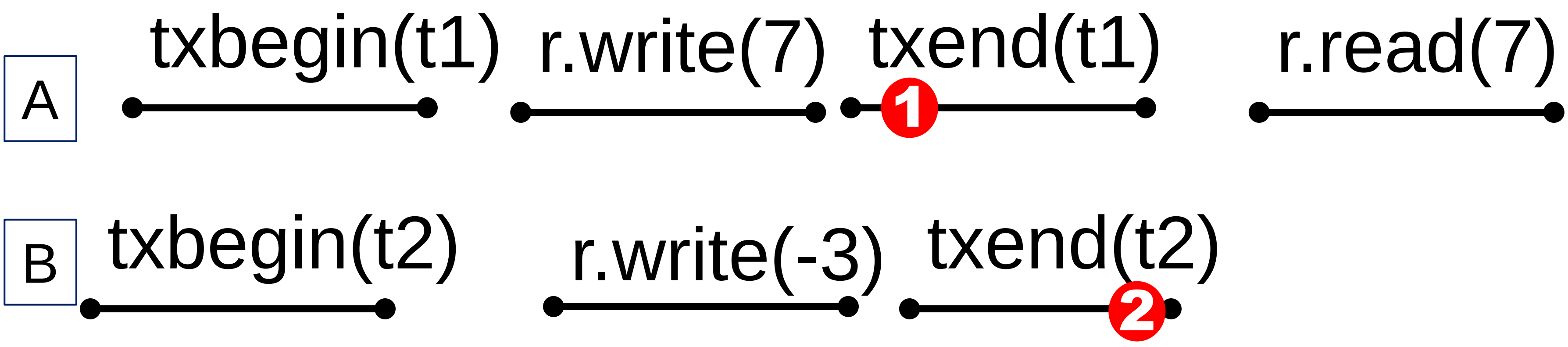


# Atomicity or Isolation?

- A thread gets exclusive access and dies
  - For atomicity, abort and roll back transaction
  - For isolation, B cannot block indefinitely because of A, so transaction must abort



# Durability or Isolation?




- Last read should be -3
  - Might be a durability failure
  - Might be a isolation failure
- Resultant history looks bad
  - Not sequentially consistent

# Let’s retire ACID

- Transactions have AID, not ACID
- Atomicity, isolation, and durability are poorly differentiated
  - Real situations are a superposition
  - Distinction makes you see things that aren’t there
  - Subsumed by schedules

# Where do we go from here?

- Concurrent operations need to be scheduled – TPS
  - Traditional scheduling via locking
  - Performance issues
- Generalize the notion of transaction and transaction processing system.
  - TPS: seq. consistency, linearizability, dependent transactions
  - Read-copy update: Radical future

# Designing a TPS

- TPS schedules operations
- Operations have defined semantics
  - E.g., read returns last written value
  - Constrains correct executions
- Figuring out new scheduling models and/or new operations ongoing work
  - E.g., read_best_effort()

# Basic algorithm for serializability

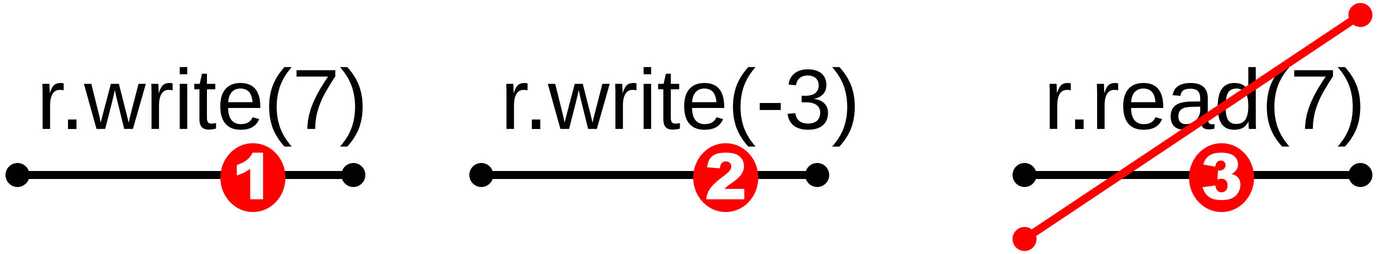
- Before reading data, acquire its read lock
- Before writing data, acquire its write lock
- Before searching (updating) a predicate, acquire a read (write) lock on the predicate (DB only)
  - Protects both real and (infinite) phantom items

- If locks from two transactions conflict, abort one
  - Locks conflict if at least one is a write lock
- Hold all locks until transaction commit
  - 2 phase locking (acquire and release phases)

# Concurrency and performance

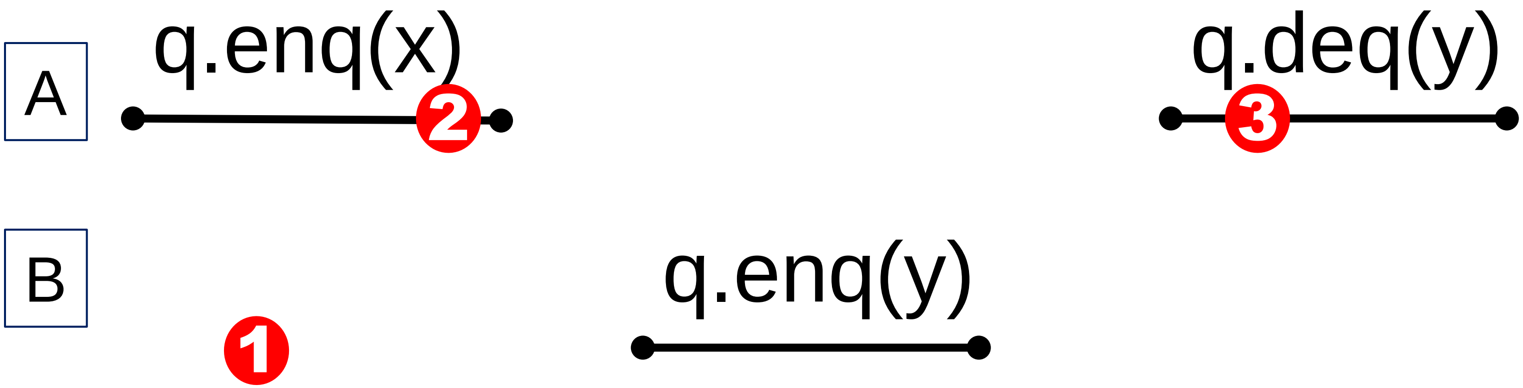
- More legal schedules = More performance
  - More concurrency
  - More scalability
  - Two phase locking often lacks performance
- Weak semantics = More schedules
  - E.g., item appears to be on list twice
  - Weak semantics = programming difficulty
    - Try eventual consistency for distributed systems

# Sequential consistency



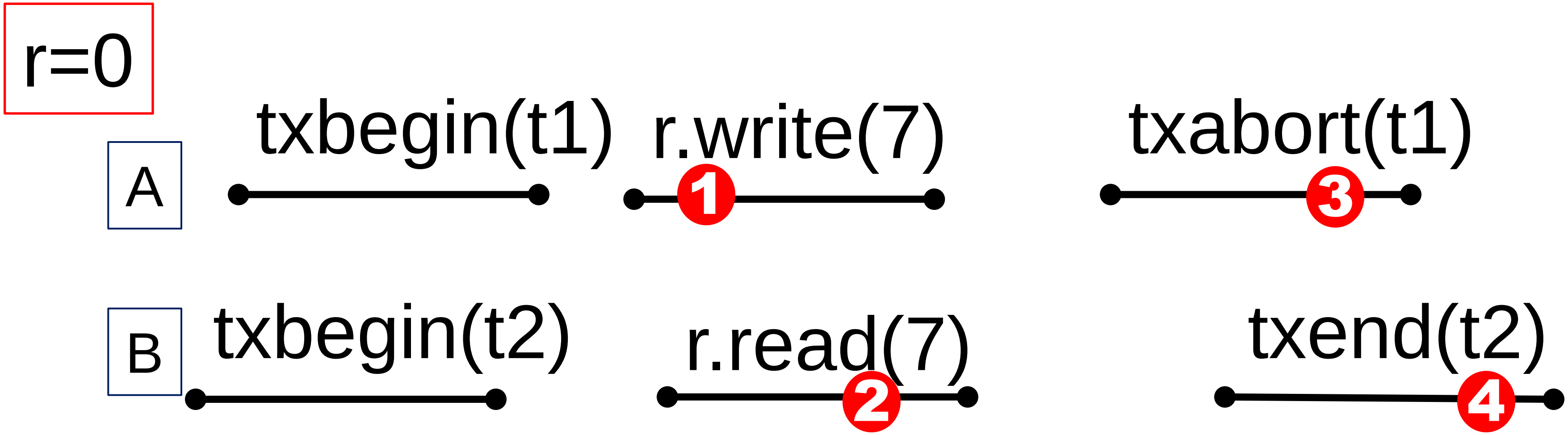

- Sequential consistency used in multiprocessors
  - Methods appear one-at-a-time, sequentially
  - Methods must appear in program order
- read(7) is not sequentially consistent
  - Though legal for weaker models

# Linearizability



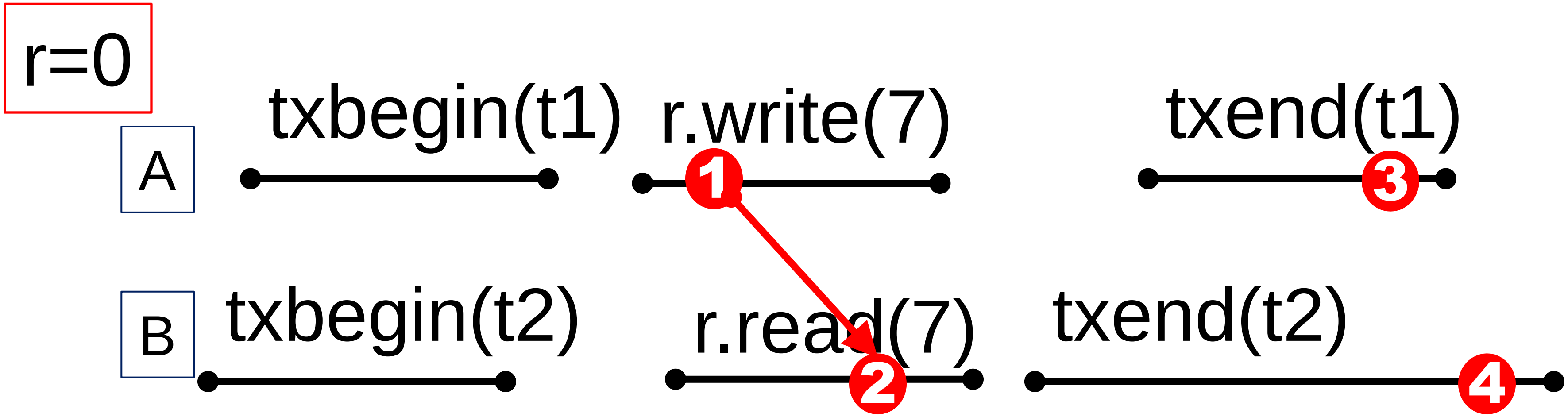

- FIFO queue
- History is serializable, but does not respect real time order
- Sequentially consistent, not linearizable

# Classic isolation failure




- Data written by t1 read by t2 (dirty read)
- t2 commits!
- Where did read data come from?

# Dependent transactions

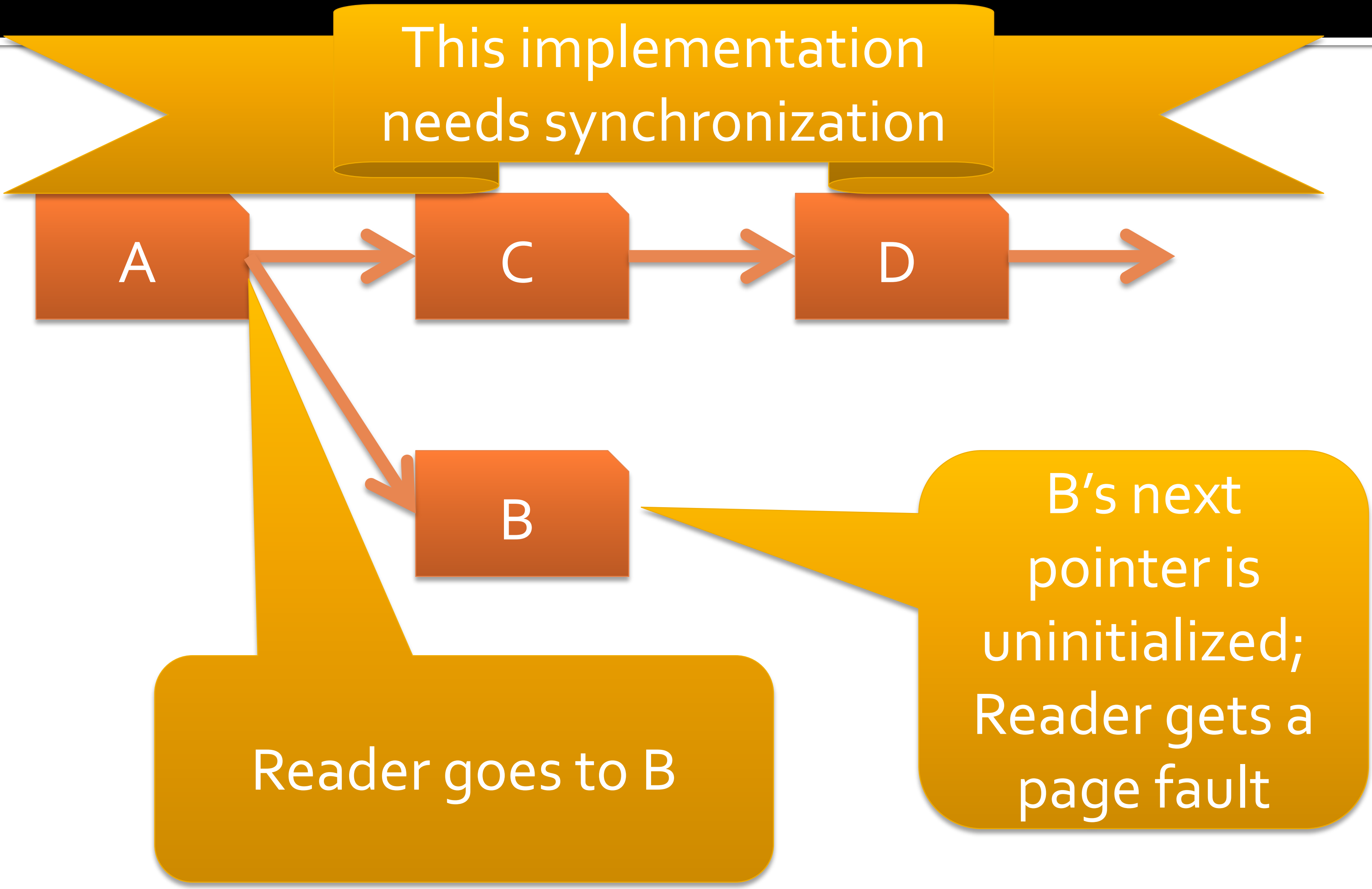



- Data written by t1 forwarded to t2
  - t1 must commit before t2
  - If t1 aborts, t2 must abort (no dirty read)
- TPS accepts more schedules
  - Cascading aborts? Only problem for DB systems

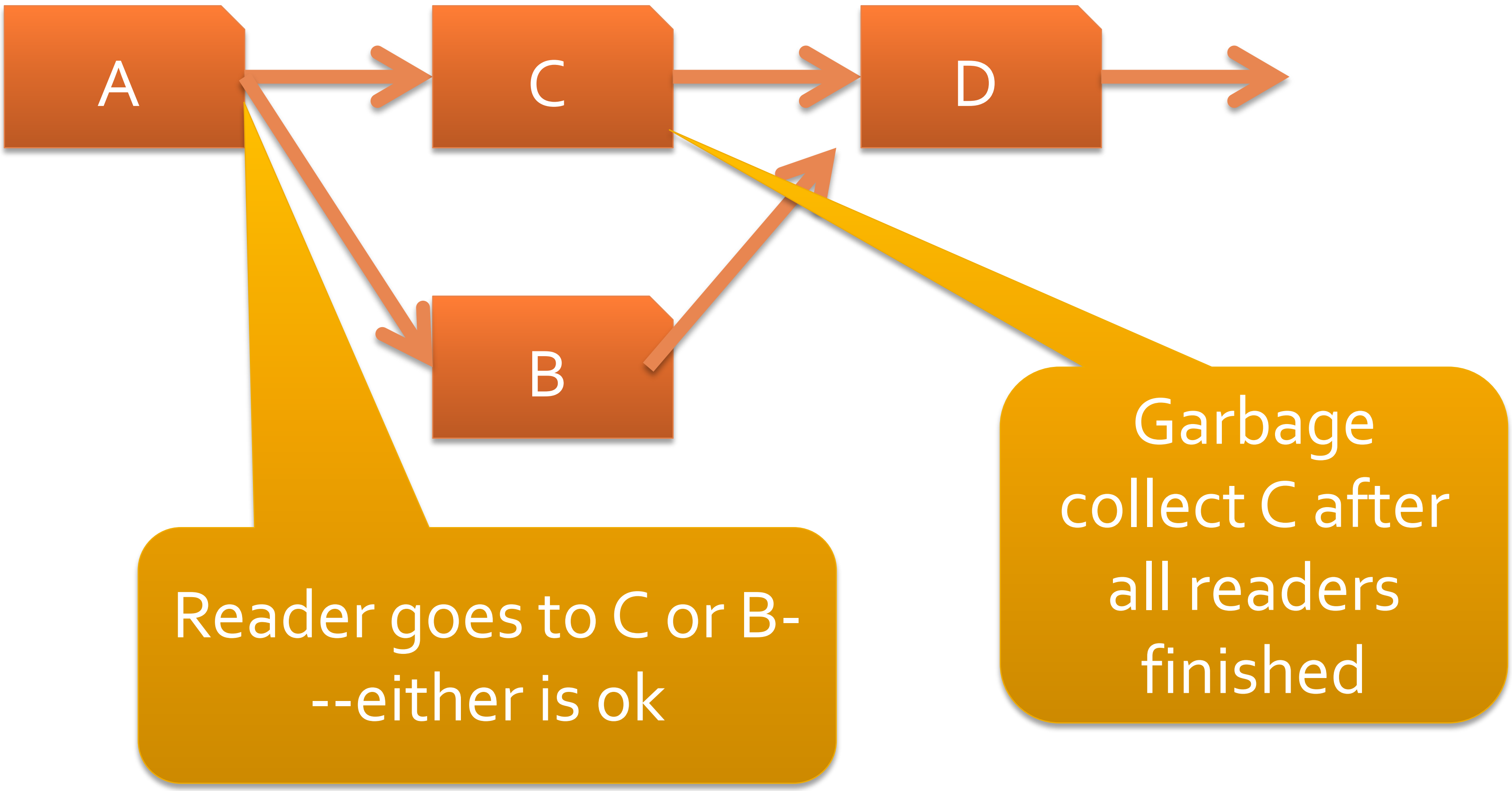
# Read-copy update (RCU)

- Defines readers and writers
  - Begin read-only transaction
  - More like reader-writer lock than transaction
- Reduce read synchronization to nothing
  - Avoids expensive atomic instructions & fences
- Make writers careful
  - Readers always see a consistent view
- Specialized to lists (but that is changing)

# Example: Linked lists



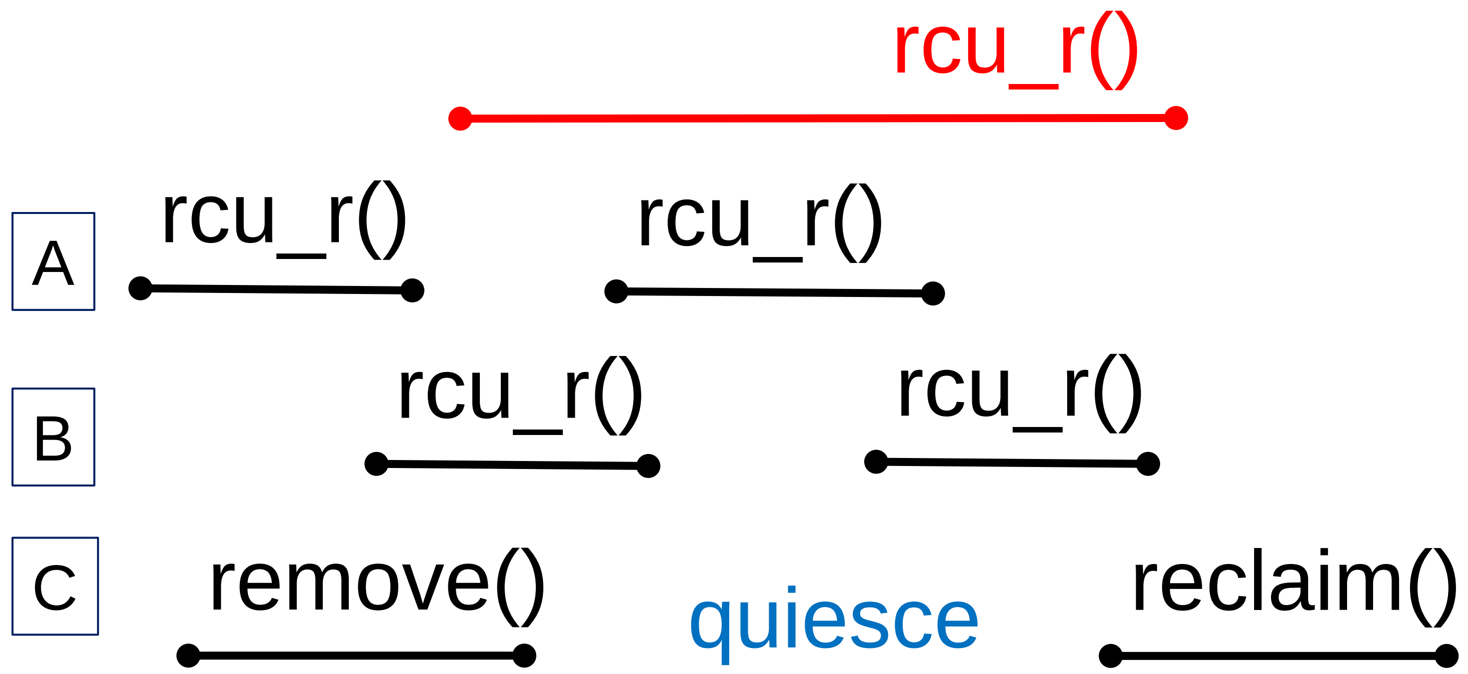
# Example: Linked lists



# Basic RCU lists [McKenney]

- Create node B, with all outgoing pointers
- Then overwrite the pointer from A
  - Either traversal is safe
  - No atomic instruction needed
  - Need compiler memory barrier
    - HW memory barrier only on DEC Alpha
- List always readable
  - Writers must take care
  - Writers might wait for all current readers (quiesce)

# Scheduling RCU




- Remove item: pointer write
- Reclaim: memory free
- TPS lengthens quiescence period as needed

# Feel the power

- Exercise: Describe RCU with ACID
  - Heck, describe RCU
- Generalizing transactions and TPS
  - Databases
  - Transactional memory
  - Distributed systems

# Searching for meaning in concurrency

# My group's work

- TxLinux & MetaTM [ISCA, SOSP '07, CACM '08]
  - Transactions if possible, locks when necessary (I/O)
- Dependent transactions [MICRO '08, PPoPP '09]
  - Committing conflicting transactions
- Synchronization in Linux [HotOS '07, ISPASS '10]
  - Will optimistic primitives scale? Data independence
- HW, SW coordinated transactions [ASPLOS '09]
- OS transactions [SOSP '09, Eurosys '12]
- Thanks to: Hany E. Ramadan, Christopher J. Rossbach, Indrajit Roy, Donald E. Porter, Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, Mark Silberstein, Yuanzhong Xu

# Reading list

- The Transaction Concept: Virtues and Limitations [Jim Gray 1981 IEEE]
- Principles of Transaction-Oriented Database Recovery [Haerder & Reuter 1983 ACM]
- Linearizability: a correctness condition for concurrent objects [Wing & Herlihy 1990 TOPLAS]
- Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial [Fred Schneider 1990 ACM]
- Transaction Processing [Gray and Reuter 93 MK]
- *A Critique of ANSI SQL Isolation Levels [Berenson, Bernstein, Gray, Melton, O’Neil, O’Neil 1995 MSR-TR]
- *The Art of Multiprocessor Programming [Herlihy & Shavit 2008]
- Principles of Transaction Processing [Bernstein & Newcomer 2009 MK]

# Insight

# Conclusions

- Concurrency management is fun
  - Great need for progress
  - Ample opportunities for progress
- Don’t use ACID as a crutch
- Schedule concurrency
  - Search for meaning