

Consistency, Transactions

Amin Vahdat

CSE 123b

May 30, 2006

Annoucements

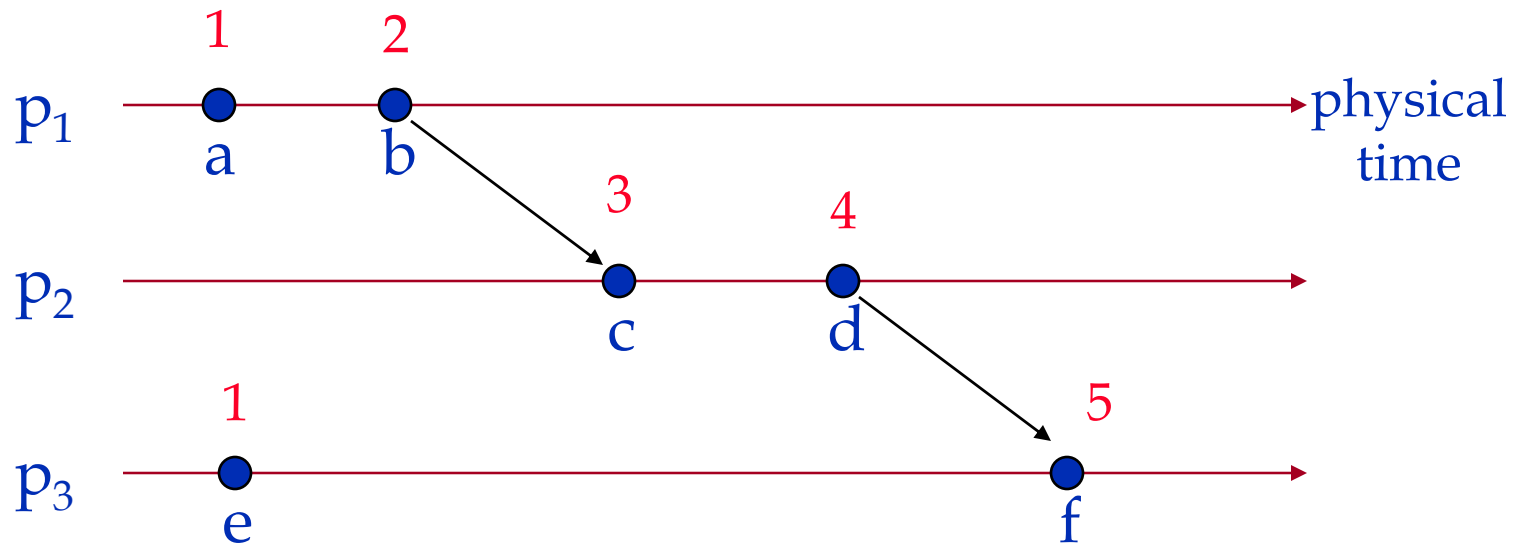
- Third assignment
 - Due date June 9, 5 pm
- Final exam, June 14, 11:30-2:30

Eventual Consistency through Anti-Entropy

Problem Scenario

- Replicated Service
 - Perhaps across the Internet, perhaps in mobile computing environment
- Clients can apply updates to any replica (read any replica)
- Replicas may never be fully connected
 - How to achieve eventual consistency?

Logical Clocks



- Assign a monotonically increasing logical “timestamp” to all updates
- On message exchange, set local logical clock to be “one more” than maximum logical clock value among hosts

Consistency through Pair-wise Message Exchange

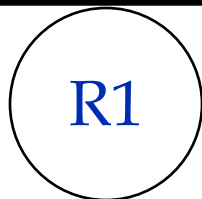
- Each replica maintains *logical time vector* of length n
 - Where n is the number of replicas
 - Entry $ltv_j[i]$ is the last known logical time for server i known by server j
- Periodically, a pair of replicas come into contact with one another → exchange messages
 - Analogous to gossip exchange
 - Logical time vector states which updates the replicas are missing relative to one another

Exchange necessary updates

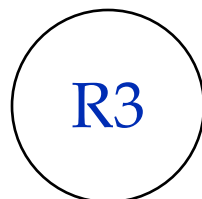
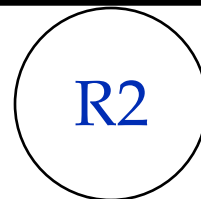
Update logical time vectors

Anti-Entropy: Simple Example

6	0	0
---	---	---



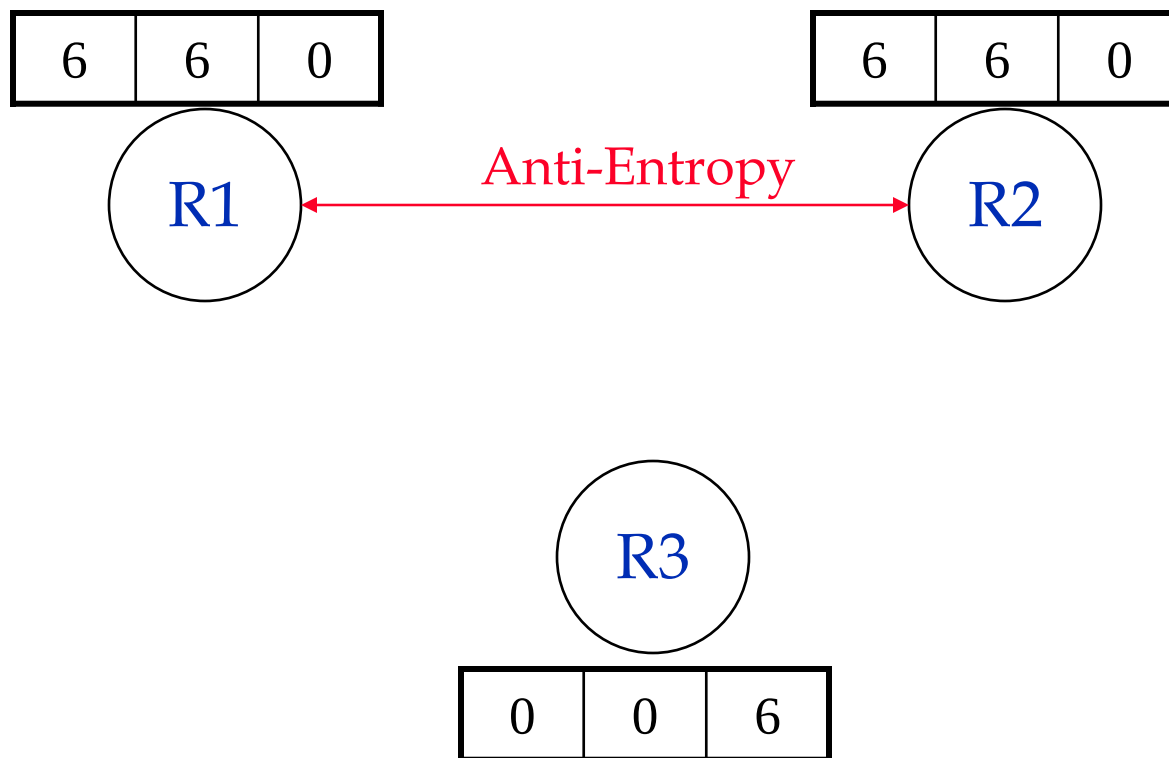
0	6	0
---	---	---



0	0	6
---	---	---

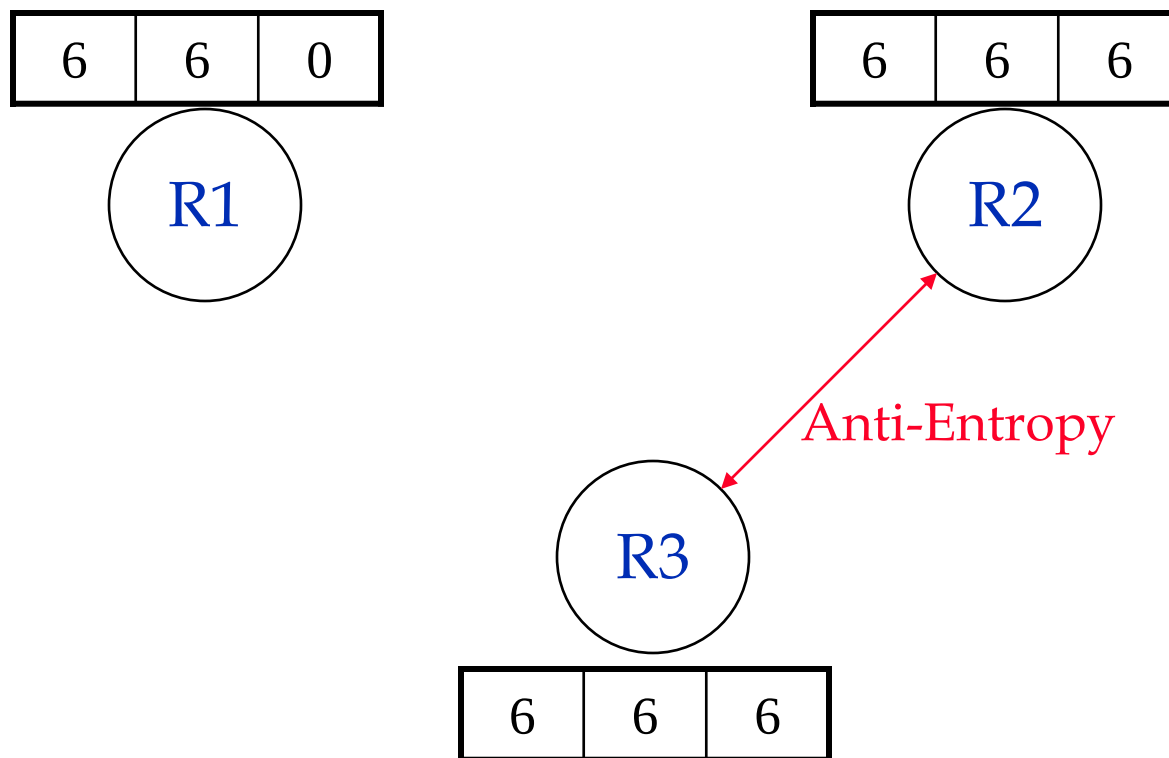
- Each of R1, R2, and R3 have seen 5 different updates (for total of 15 independent updates)

Anti-Entropy: Simple Example



- R1 and R2 perform anti-entropy
 - R1, R2 have 10 updates, R3 has 5 updates

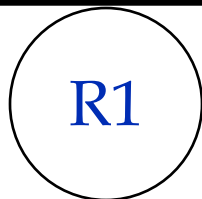
Anti-Entropy: Simple Example



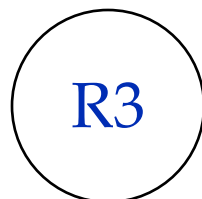
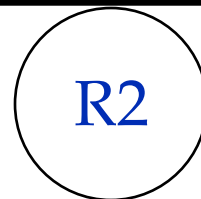
- R2 and R3 perform anti-entropy, has 15 updates
 - Its own updates at times 1-5, 10 remote updates at time 6

Anti-Entropy: Simple Example

6	6	0
---	---	---



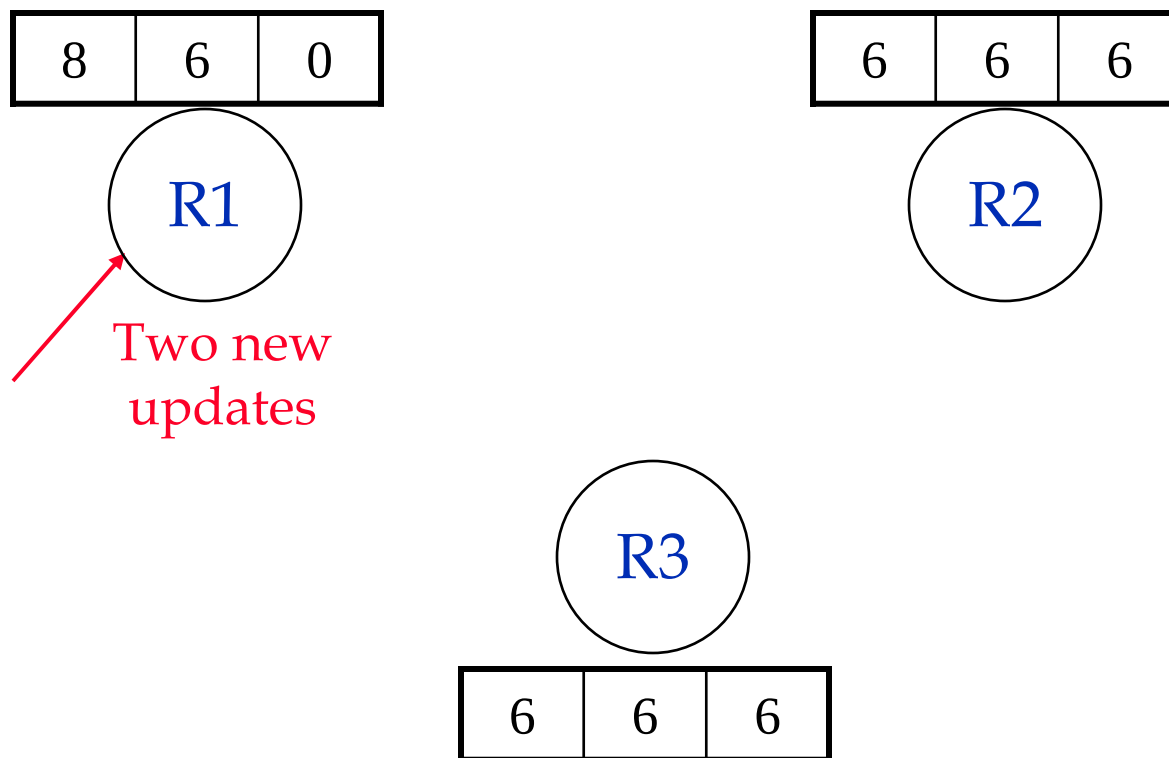
6	6	6
---	---	---



6	6	6
---	---	---

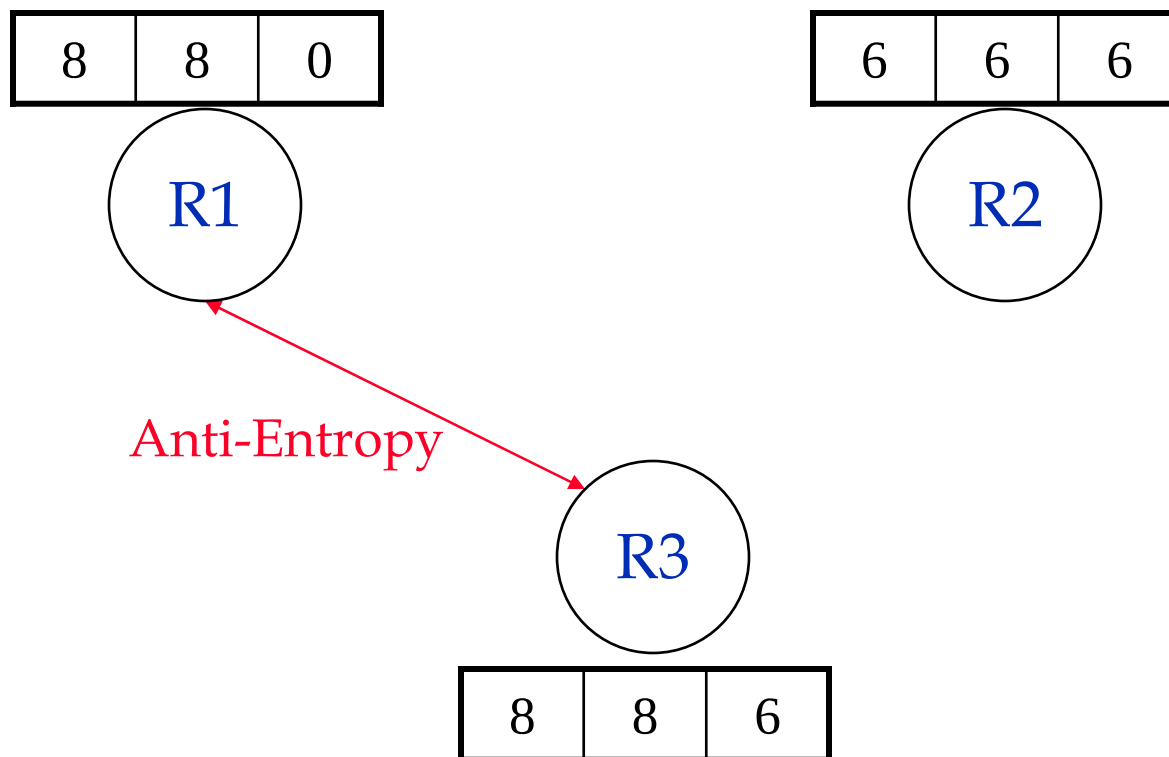
- What happens when R1 and R3 perform anti-entropy?
 - What if R1 accepts two new updates in the interim?

Anti-Entropy: Simple Example



- On Anti-Entropy between R1 and R3, R1 only propagates writes between time 6 and 8, R3 propagates all writes (after 0)

Anti-Entropy: Simple Example



- Anti-Entropy maintains causal order (happens-before) but (as described) does not impose a total order

Transactions

ACID Property

- Atomicity
 - All or nothing (even in the face of failure)
- Consistency
 - Take system from one consistent state to another
- Isolation
 - No interference from concurrent transactions: do not see the intermediate (uncommitted) results of any transaction
- Durability
 - Once *committed* all effects are saved in permanent storage

Serial Equivalence

- One approach to avoiding concurrency problems is to execute transactions in *serial* order
 - Do not begin transaction $n+1$ until transaction n either commits or aborts
 - Results in unacceptable performance
 - What if transactions are not accessing the same data items?
- Another approach is to ensure that transactions execute in *serial equivalent* order
 - Interleave operations only when it is safe to do so
 - Final result same as if transactions performed in serial order
- How to achieve serial equivalence?

Achieving Serial Equivalence

- Approaches to serial equivalence
 - Locking
 - Optimistic concurrency control (check on commit)
 - Timestamps
- Achieve serial equivalence by locking necessary data items
 - Avoiding deadlock
 - Breaking deadlock
 - Granularity of locks
 - Read/write (shared/exclusive) locks
 - Two phase commit for distributed lock management

Two-Phase Locking

- Serial equivalence requires that all data accesses be serialized w/respect to other transactions
 - If two transactions conflict, then *all* operations within transactions must be (logically) done either before or after other transaction
- Two-phase locking: do not acquire any new locks after releasing any lock
 - Growing phase, followed by shrinking phase
- Strict two-phase locking
 - Must worry about aborted transactions (dirty writes)
 - Do not release locks until transaction aborts/commits

Lock Manager

- Want to automate the process of acquiring/releasing locks
 - Avoid forcing programmers to get it right every time
 - Use API to force all data accesses go through lock manager
- Inside transaction, on data access:
 - If object not locked, server locks it
 - If conflicting lock, block transaction (Wait on condition var)
 - If non-conflicting lock, acquire shared lock
 - For read/write access consider lock promotion
- On transaction commit/abort:
 - Release all locks, signal appropriate waiting transactions
 - Starvation?

Deadlocks

- Deadlock occurs when a loop is created in a logical *waits for* graph
 - Transaction T acquires A, Transaction U acquires B
 - Transaction T waits for B, Transaction U waits for A
 - Neither transaction able to make forward progress
- Loop can be arbitrarily long
 - $T \rightarrow U \rightarrow V \rightarrow W \rightarrow \dots \rightarrow Z \rightarrow T$
 - Typically, loops are short (one hop)

Deadlock Example

Transaction T Withdraw(A, 4); Deposit(B, 4)	Transaction U Withdraw(C, 3); Deposit(B, 3)
time Balance=A.Read() Read A A.Write(balance-4) Balance=B.read() Read B	Balance = C.Read() Read C C.Write(balance-3) Balance=B.Read() Read B

Deadlock Example

Transaction T Withdraw(A, 4); Deposit(B, 4)	Transaction U Withdraw(C, 3); Deposit(B, 3)
Balance=A.Read() A.Write(balance-4) Read A Balance=B.read() B.Write(balance+4) Read B Wait U	Balance = C.Read() C.Write(balance-3) Read C Balance=B.Read() Read B

time



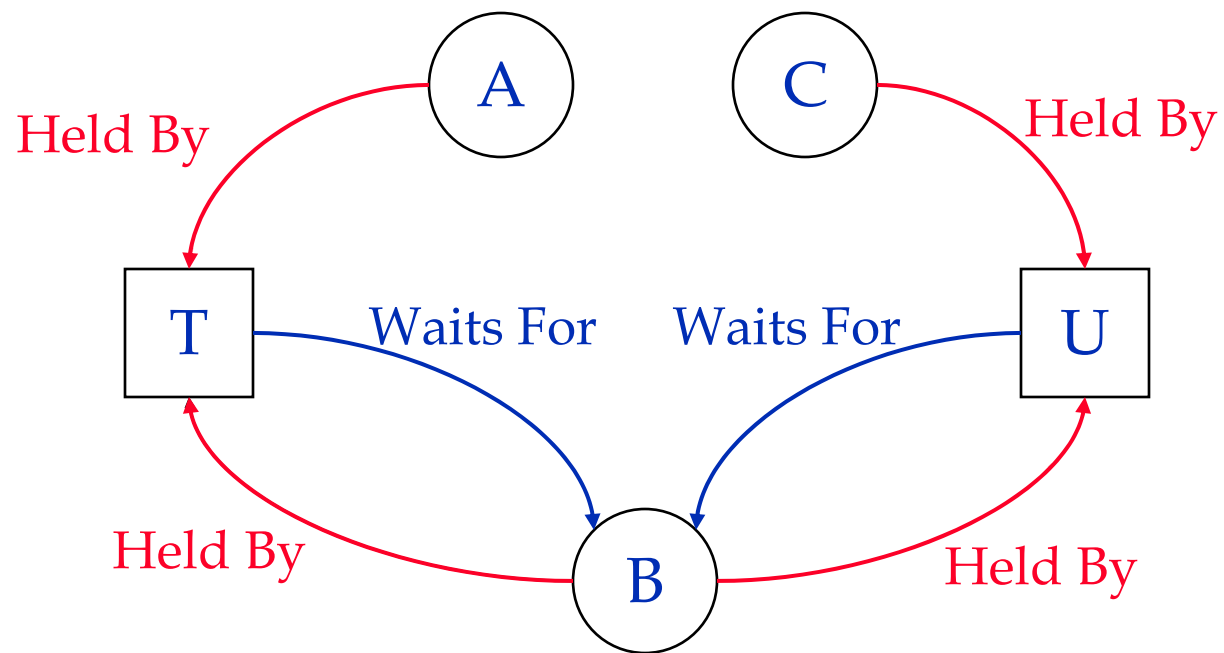
Deadlock Example

Transaction T Withdraw(A, 4); Deposit(B, 4)	Transaction U Withdraw(C, 3); Deposit(B, 3)
Balance=A.Read() Read A A.Write(balance-4)	
	Balance = C.Read() Read C C.Write(balance-3)
Balance=B.read() Read B	Balance=B.Read() Read B
B.Write(balance+4) Wait U ...	B.Write(balance+3) Wait T ...
...	

time



Waits-For Graph



- Cycle: $T \rightarrow B \rightarrow U \rightarrow B \rightarrow T$

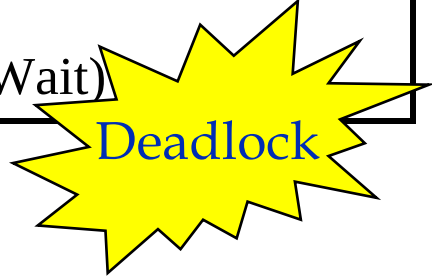
Deadlock Prevention

- Gather *all* locks at beginning of transaction
 - Can we still run into problems?

Deadlock Prevention

- Gather *all* locks at beginning of transaction
 - Can we still run into problems? Yes, if locks are acquired in different order

Transaction T	Transaction U
Lock A	Lock B
Lock B (Wait)	Lock A (Wait)



Deadlock

Deadlock Prevention

- Gather *all* locks at beginning of transaction
 - Can we still run into problems? Yes, if locks are acquired in different order
 - Solution: gather locks in canonical order
- Won't work in many circumstances
 - Cannot predict locking requirements of interactive apps
 - Unnecessarily reduces concurrency
 - E.g., acquire read (write) lock for long-running transaction
 - Prevent *any* other transaction from acquiring write (read) lock

Deadlock Detection

- Concurrent thread keeps track of waits-for graph
 - Typically lock manager
 - For each successful lock operation, track *resources* held by each *transaction*
 - Not strictly necessary
 - For each conflicting lock operation (condition var wait), track transaction *waits for* relationship
 - On lock release, delete edges corresponding to *signaled* transactions

Deadlock Detection

- Search for loops in *waits for* graph on adding edge
 - Abort one transaction in the loop
 - Release all locks associated with transaction

Releasing locks signals blocked transaction in loop, break deadlock
 - Aborted transaction must restart

Starvation
- Which transaction to abort?

Deadlock Detection

- Search for loops in *waits for* graph on adding edge
 - Abort one transaction in the loop
 - Release all locks associated with transaction
 - Releasing locks signals blocked transaction in loop, break deadlock
 - Aborted transaction must restart
 - Starvation
- Which transaction to abort?
 - Assign priorities to transactions?
 - Shortest running transaction?

Preventing Deadlocks Through Timeouts

- Assign a timeout with each lock
 - When timeout expires, lock becomes *vulnerable*
- If no other transaction waiting for lock, continue as normal
- If other transaction waiting, abort transaction holding vulnerable lock
- Pros/cons?

Preventing Deadlocks Through Timeouts

- Assign a timeout with each lock
 - When timeout expires, lock becomes *vulnerable*
- If no other transaction waiting for lock, continue as normal
- If other transaction waiting, abort transaction holding vulnerable lock
- Pros/cons
 - Pros: simpler to implement?
 - Cons: abort transaction when no deadlock exists, what value to set for timeout (lightly loaded vs. heavily loaded system)