Recap

- Two-Phase Locking (2PL)
- Intro to Transactions

Today

- Transactions
- Concurrency Control

Transactions

- Concurrency control through mechanisms such as 2PL is an important component of *transactions*
- Mutual exclusion algorithms ensure that a set of shared resources is only used by one process at a time
- Transactions go further: the operations performed on multiple shared resources are performed as one atomic operation
- If a transaction doesn't finish successfully, all resources are restored to the states they had before the transaction started

Transaction Model

- Processes = signatories to a contract
- One process (the *initiator*) starts a transaction with one or more other processes, and various actions take place “within” the transaction
- At some point, the initiator decides that the transaction should be *committed*
  - If the other processes agree, the actions of the transaction become permanent (atomically)
  - If not, the system returns to the state it had before the transaction

Transaction Model: Example

- Assume a customer wants to transfer money between two of their accounts at the same bank
- Essentially a *withdrawal* followed by a *deposit* - but it's critical that both are completed successfully
  - If the system crashes after the withdrawal, but before the deposit, the withdrawn money vanishes into the ether…
- If the withdrawal and deposit are grouped in a transaction, we don't have to worry about vanishing funds (due to partial failures)

Transaction Primitives

- Primitive transaction operations that exist in (essentially) all transaction systems:
  - *BEGIN_TRANSACTION* - starts a transaction
  - *END_TRANSACTION* - ends a transaction and tries to commit its results
  - *ABORT_TRANSACTION* - ends a transaction and restores the system to its pre-transaction state
- Operations actually performed inside transactions are system-dependent (such as *READ*, *WRITE*, *DEPOSIT*, *WITHDRAW*, *SEND*, *RECEIVE*, etc…)

Example Revisited

- How would a transfer of *x* dollars from account *a* to account *b* look with transaction primitives?
  - Assume that WITHDRAW and DEPOSIT return Boolean values to indicate success or failure

```
BEGIN_TRANSACTION



END_TRANSACTION
```

Example Revisited

- How would a transfer of *x* dollars from account *a* to account *b* look with transaction primitives?
  - Assume that WITHDRAW and DEPOSIT return Boolean values to indicate success or failure

```
BEGIN_TRANSACTION
success = WITHDRAW x, a
success = success AND (DEPOSIT x, b)
IF (!success) THEN ABORT_TRANSACTION ELSE
END_TRANSACTION
```

ACID Properties

- Transactions have several properties, not just the ones we've already mentioned
  1) *Atomicity*
  2) *Consistency*
  3) *Isolation* (which you know as *Serializability*)
  4) *Durability*
- These are commonly known as the *ACID properties*, from their initial letters

Atomicity

- Either a transaction happens completely, or it does not happen at all
- If a transaction happens, it appears to happen as a single indivisible action
- When a transaction is in progress, processes not involved in it cannot see the intermediate states that occur
  - This includes processes involved in other concurrent transactions

Consistency

- If the system has certain invariants, then if they held before a transaction, they hold after the completion of that transaction
  - In banking systems, a system invariant is that after any internal transfer, the amount of money in the bank must be the same as it was before the transfer - the law of conservation of money
- That doesn’t mean that the invariants need to be true throughout the transaction, though (since the intermediate states are not observable)

Isolation (Serializability)

- If two or more transactions are running concurrently, then to each of them and to all external processes, the final result looks as though all transactions ran sequentially in some (system-dependent) order
  - That is, they appear to have run *serially*, rather than in parallel

Durability

- Once a transaction is successfully committed, regardless of what happens afterward, its results become permanent
- Specifically, no failure after the commit can undo the results or cause them to be lost
- Clearly, special algorithms are necessary to ensure this… (fault tolerance)

Classification of Transactions

- A sequence of operations that satisfies the ACID properties is called a *flat transaction*
- There are many limitations to flat transactions, and many alternative transaction models have been discussed in the literature
- *Nested transactions* and *distributed transactions* are two other important classes of transactions

Nested Transactions

- A nested transaction is constructed from a set of *subtransactions*
- These subtransactions may run in parallel to improve efficiency or simplify programming
- Each subtransaction may also have subtransactions, and nesting can occur to arbitrary depth
- What issues arise from nesting?

Nested Transactions

- Suppose a subtransaction commits, but the parent transaction subsequently aborts - the subtransaction’s actions must be undone to preserve the atomicity of the top-level transaction
- For this to work, durability can only apply to the top-level transaction and not to subtransactions
- Implementation can be tricky, but the required semantics are fairly straightforward - suggestions?

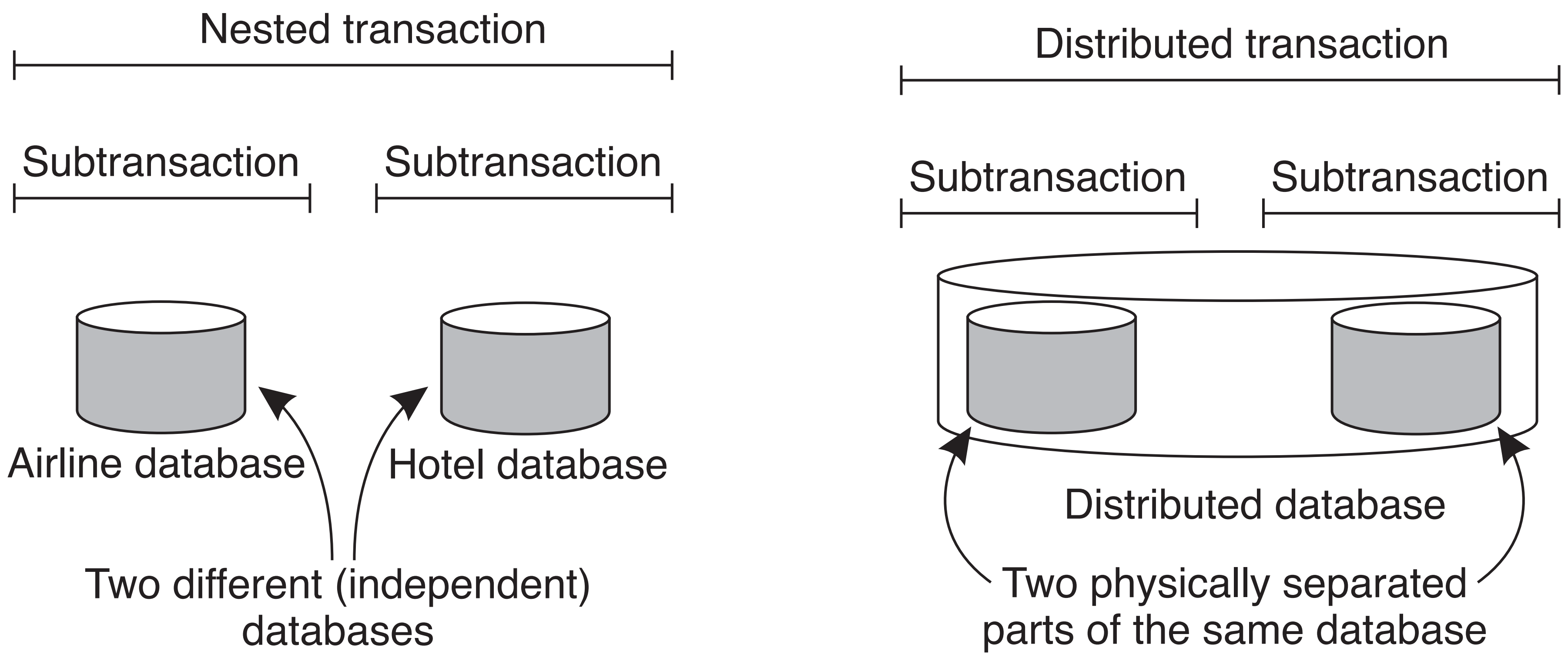
Semantics of Nested Transactions

- When a subtransaction starts, it is given its own private copy of the system (universe) to modify as it will
- When a subtransaction commits, its private universe replaces its parent's universe
- If a new subtransaction is started, it sees the results of previously-committed subtransactions
- If a (sub)transaction aborts, all its subtransactions must also abort
- Anything missing?

Distributed Transactions

- Distributed transactions deal with a physical division of transactions caused by the need to access distributed resources
- Special distributed algorithms are needed to handle locking of data and committing of transactions

Nested vs. Distributed Transactions



Implementation of Transactions

- Now we’ve described the properties of transactions for dealing with multiple resources
- Transactions sound like a very nice thing to have…
- Just one problem - how do we implement them?

Implementation of Transactions

- Consider transactions that read and write sets of files on a (simplified) filesystem
  - Database transactions are roughly analogous
- Filesystem Structure
  - There is a set of blocks on the filesystem, and each file is composed of some sequence of these blocks
  - For each file, there is an *index* (in a directory somewhere) of the blocks contained therein
  - Every operation either reads from a file or writes to a file, so it reads or writes one or more blocks
- Any suggestions about transaction implementation? (think big picture, not block level just yet…)

Two Implementation Methods

- Private Workspace
  - Each transaction essentially plays in its own copy of the universe
- Writeahead Log
  - Changes are logged so that, if a transaction aborts, the changes can be undone

Private Workspace

- When a process starts a transaction, it is given a private workspace containing all the files it can access
- Until the transaction commits, all reads and writes go to this private workspace instead of going directly to the filesystem

Private Workspace

- Clearly, we can’t afford to *really* copy all the files into a private workspace for each transaction, if only because of memory and disk space constraints
- Need to come up with some clever optimizations
  - Reads are different from writes…
  - Using the low-level (index and block) structure of the filesystem may help…

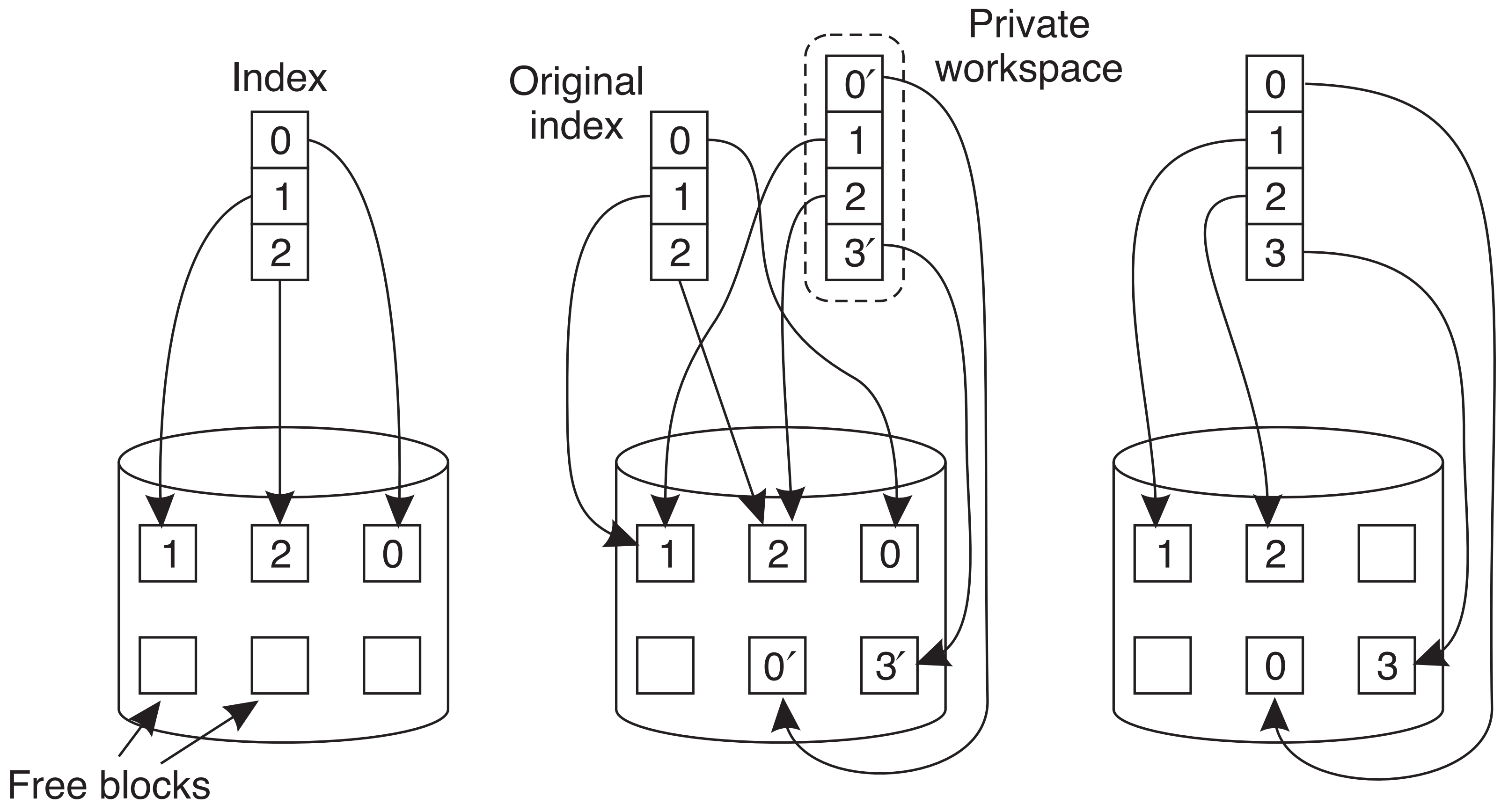
Private Workspace Optimizations
Read-Only Access

- When a process reads but doesn't modify a file, it doesn't need its own copy of that file unless the file changes after the transaction has started
- We can create a private workspace that refers to a parent workspace (which could be the actual filesystem), as long as we make private copies when files change
- This gives us access to all the files with the states they had at the beginning of the transaction
- When the transaction ends, we can just throw away the private workspace (if the process has only done reads)

Private Workspace Optimizations Read-Write Access

- When a file is opened for writing, we copy only its index (the information used by the filesystem to track the disk blocks that are part of the file) to our private workspace
- If we change the file, we make a private copy of the disk block we changed (this is called a *shadow block*) and change our index to point to it - this works for adding blocks also
- If the transaction commits, we replace our parent's index with our index
- If the transaction aborts, we just throw our index away

Private Workspace Optimizations Read-Write Access



Private Workspace Distributed Transactions

- How would this work on a distributed filesystem?
- We haven’t really talked about distributed filesystems - the general idea: multiple machines have individual filesystems that are combined to form a single large filesystem
- Assume that the filesystem is distributed such that each file resides on a single machine (and therefore on that machine’s local filesystem), rather than that a file can be split over multiple machines
- Implementation suggestions?

Private Workspace Distributed Transactions

- Transaction starts a process on each machine whose filesystem contains a file used in the transaction
- Each process has a private workspace for just the files on its own machine
- If the transaction aborts, all the processes terminate and throw out their private workspaces
- If the transaction commits, all changes are made locally by the processes
- What can go wrong?

Writeahead Log

- All files are actually modified in place
- Before any change to a file, a record of the change is written to a log:
  - transaction identifier
  - file and block that are being changed
  - old and new values of the block
- The actual change to the file is only made after the log record has been successfully written

Writeahead Log Example

```
x = 0;
y = 0;
BEGIN_TRANSACTION;
 (1) x = x + 1;
 (2) y = y + 2;
 (3) x = y * y;
END_TRANSACTION;
```

```
(1) [x = 0/1]

(2) [x = 0/1]
    [y = 0/2]

(3) [x = 0/1]
    [y = 0/2]
    [x = 1/4]
```

Writeahead Log

- If the transaction succeeds and is committed, a commit record is written to the log - but since all the changes have already been made, no additional work is necessary
- If the transaction aborts, the log can be used to return the system to the original state by starting at the end of the log and working backwards - this is called a *rollback*
- How would distributed transactions work with a writeahead log?

Writeahead Log and Distributed Transactions

- In distributed transactions, each machine keeps its own log of changes to its local filesystem
- Rolling back then requires that each machine roll back separately
- Other than that, it’s identical to the single-machine case
- Again, things can go wrong…

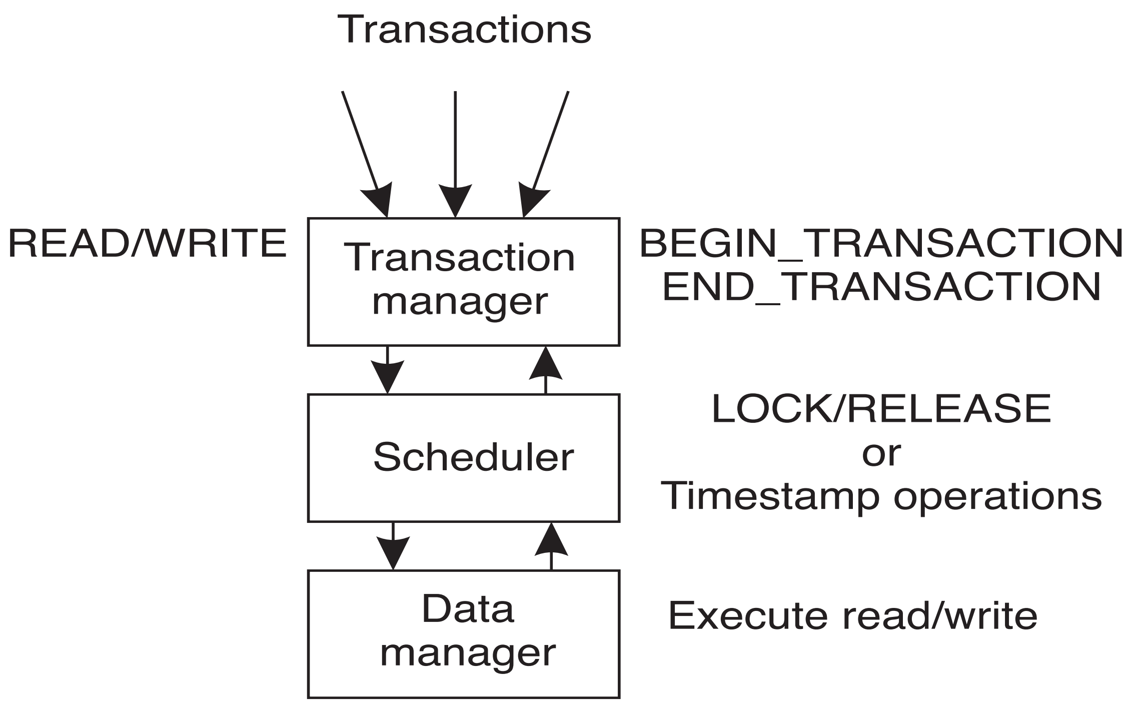
Concurrency Control

- We’ve discussed atomicity now, but what about isolation and consistency?
- *Concurrency control* allows several transactions to run simultaneously
- Consistency and isolation are achieved by giving transactions access to resources in a particular order, so that the end result is the same as some sequential ordering of the transactions
- We’ve already discussed serializability quite a bit…

Concurrency Control

- Most good system designs are layered, and concurrency control systems are no exception - they typically have three layers:
  1) *Data manager* performs the actual read and write operations on data
  2) *Scheduler* determines which transactions are allowed to talk to the data manager, and at which times (it has the bulk of the responsibility)
  3) *Transaction manager* guarantees atomicity of transactions by translating transaction primitives into requests for the scheduler

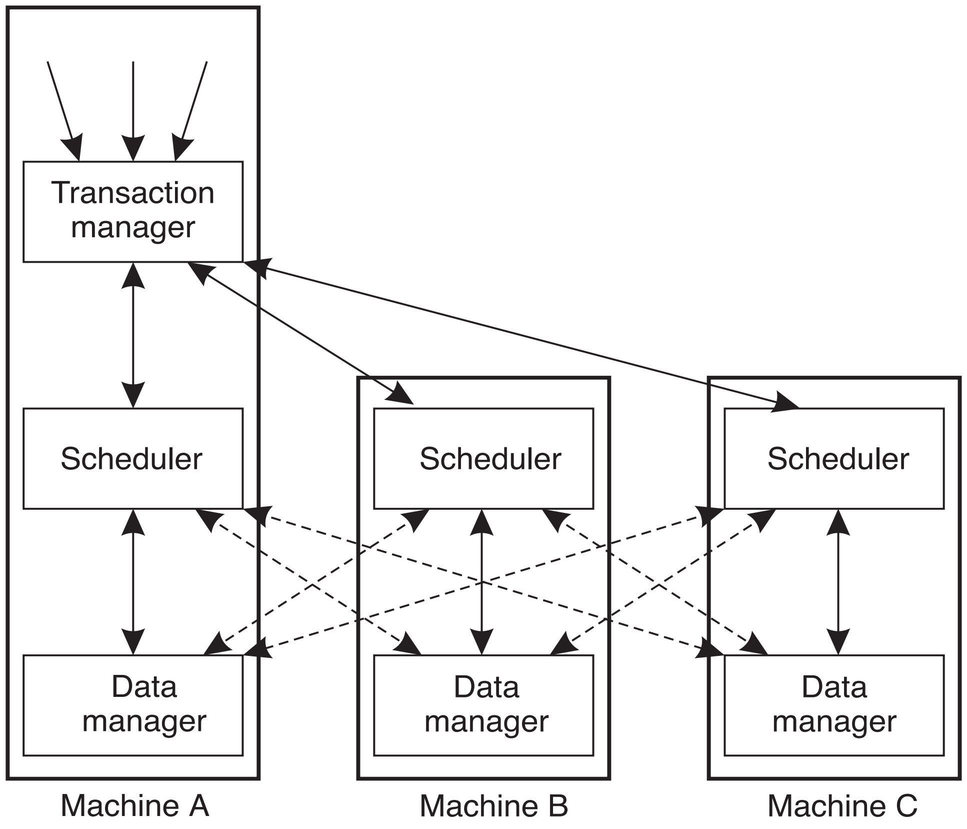
Concurrency Control



Distributed Concurrency Control

- This model can work for distributed transactions as well…
- Each machine has its own scheduler and data manager, responsible only for the local data
- Each transaction is handled by a single transaction manager, which talks to the schedulers of multiple machines
- Schedulers may also talk to remote data managers

Distributed Concurrency Control



Scheduling

- The final result of multiple concurrent transactions has to be the same as if the transactions were executed in some sequential order
- To accomplish this, we need to schedule the operations performed by the transactions in an appropriate order
- It’s not necessary to know exactly what’s being computed in order to understand scheduling - all that’s important is to avoid conflicts between operations

Scheduling Example

```
BEGIN_TRANSACTION          BEGIN_TRANSACTION          BEGIN_TRANSACTION
  x = 0;                     x = 0;                     x = 0;
  x = x + 1;                 x = x + 2;                 x = x + 3;
END_TRANSACTION            END_TRANSACTION            END_TRANSACTION
```

```
1: x = 0;  x = x + 1;  x = 0;  x = x + 2;  x = 0;  x = x + 3;

2: x = 0;  x = 0;  x = x + 1;  x = 0;  x = x + 2;  x = x + 3;

3: x = 0;  x = 0;  x = x + 1;  x = x + 2;  x = 0;  x = x + 3;
```

Scheduling Example

- Schedules 1 and 3 are legal (they both result in x being equal to 3 at the end)
- Schedule 2 is not legal (it results in x being equal to 5 at the end)
- There are a number of other legal schedules (x could be equal to 1 or 2 at the end, depending on the ordering that's decided upon for the transactions)

Scheduling

- Two operations conflict if they operate on the same data item and at least one of them is a write
  - If one of them is a write, it’s a *read-write conflict*
  - If both of them are writes, it’s a *write-write conflict*
- Two read operations can *never* conflict
- Concurrency control schemes are classified by how they synchronize read and write operations (locking, ordering via timestamps, …)

Two Scheduling Approaches

1) *Pessimistic* - if something can go wrong, it will

   Operations are explicitly synchronized before they're carried out, so that conflicts are never allowed to occur

2) *Optimistic* - in general, nothing will go wrong

   Operations are carried out and synchronization happens at the end of the transaction - if a conflict occurred, the transaction (possibly along with other transactions) is forced to abort

Locking

- Locking is the oldest, and still most widely used, form of concurrency control
- When a process needs access to a data item, it tries to acquire a lock on it - when it no longer needs the item, it releases the lock
- The scheduler’s job is to grant and release locks in a way that guarantees valid schedules
- Is locking optimistic or pessimistic?

Locking

- Locking is the oldest, and still most widely used, form of concurrency control
- When a process needs access to a data item, it tries to acquire a lock on it - when it no longer needs the item, it releases the lock
- The scheduler’s job is to grant and release locks in a way that guarantees valid schedules
- Is locking optimistic or pessimistic?

  Pessimistic

Two-Phase Locking

- We’ve already discussed 2PL from the point of view of correctness, but now we want to know how the scheduler makes decisions about scheduling read and write operations
- In 2PL, the scheduler grants all the locks during a *growing phase*, and releases them during a *shrinking phase*
- In describing the set of rules that govern the scheduler, we will refer to an operation on data item *x* by transaction *T* as *oper(T,x)*

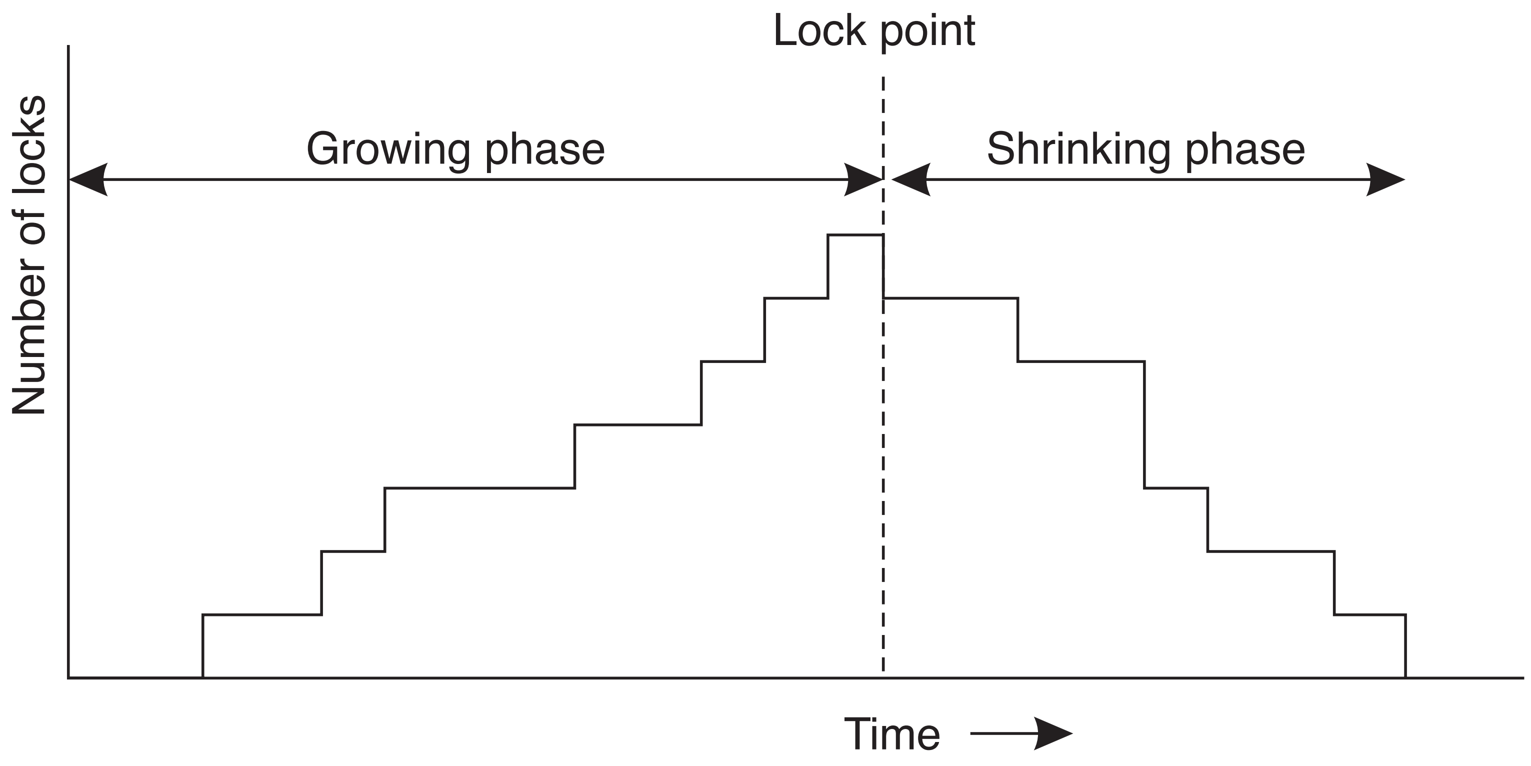
Two-Phase Locking Rules (Part 1)

- When the scheduler receives an operation *oper(T,x)*, it tests whether that operation conflicts with any operation on *x* for which it has already granted a lock
  - If it conflicts, the operation is delayed
  - If not, the scheduler grants a lock for *x* and passes the operation to the data manager
- The scheduler will never release a lock for *x* until the data manager acknowledges that it has performed the operation on *x*

Two-Phase Locking Rules (Part 2)

- Once the scheduler has released any lock on behalf of transaction *T*, it will never grant another lock on behalf of *T*, regardless of the data item *T* is requesting the lock for
- An attempt by *T* to acquire another lock after having released any lock is considered a programming error, and causes *T* to abort

Two-Phase Locking



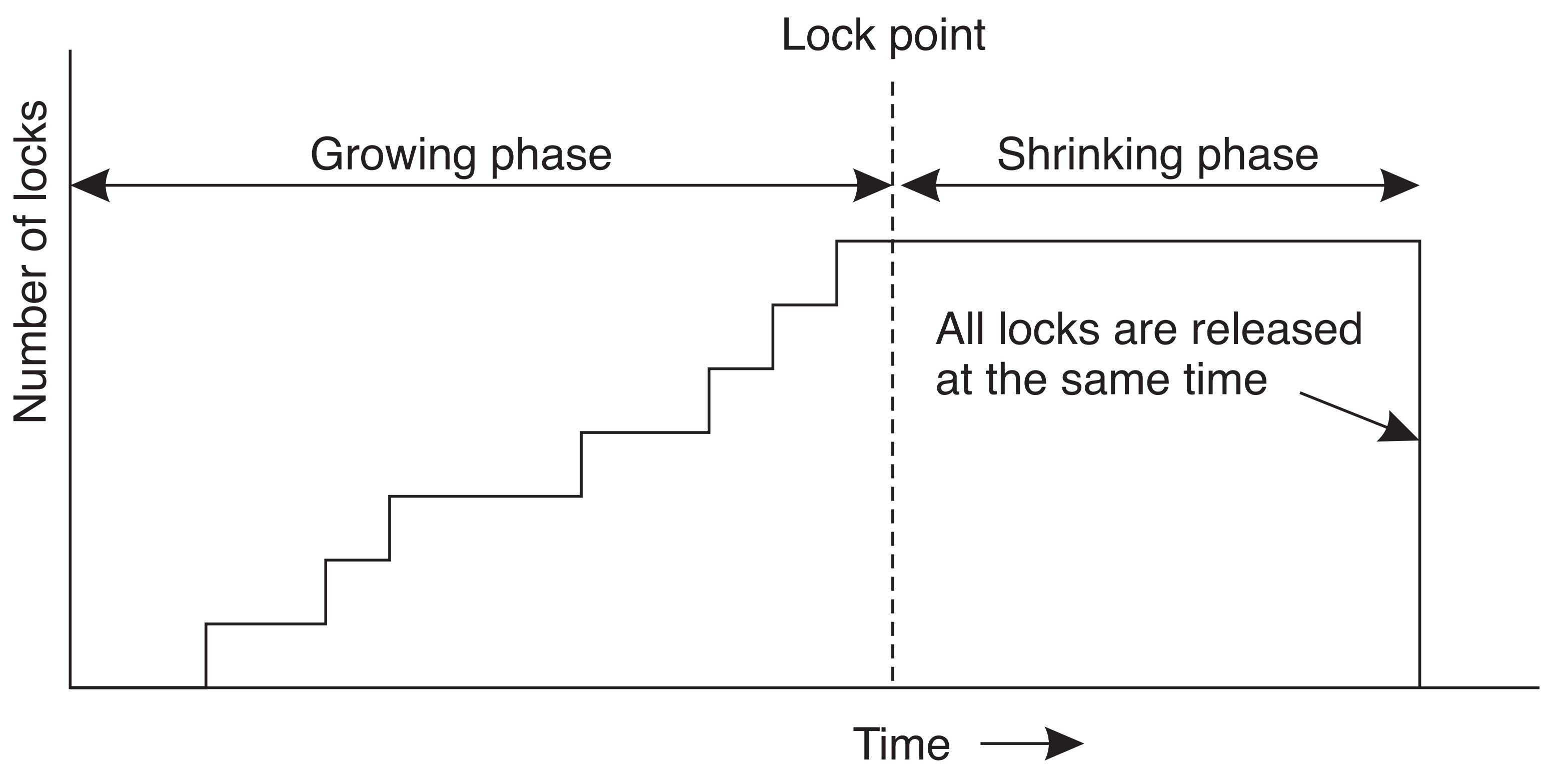
Strict Two-Phase Locking

- A variant called *strict two-phase locking* adds the restriction that the shrinking phase doesn't happen until after the transaction has committed or aborted
- This makes it unnecessary to abort transactions because they saw data items they should not have
- It also means that lock acquisitions and releases can all be handled transparently
  - How?

Strict Two-Phase Locking

- A variant called *strict two-phase locking* adds the restriction that the shrinking phase doesn't happen until after the transaction has committed or aborted
- This makes it unnecessary to abort transactions because they saw data items they should not have
- It also means that lock acquisitions and releases can all be handled transparently
  - How? Locks are acquired when data items are accessed for the first time, and released when the transaction ends

Strict Two-Phase Locking



Two-Phase Locking

- Deadlocks are possible with both two-phase locking schemes
- How?
- How do we avoid them?

Two-Phase Locking Deadlock Avoidance

- Enforce the fact that locks always need to be obtained in a canonical order (if two processes both need locks on A and B, they both request first A and then B)
- Maintain a graph of which processes have and want which locks and check for cycles, then break the cycles by aborting transactions
- Use timeouts to detect when a lock has been held for too long by a particular transaction, and abort the transaction

- The first one prevents deadlocks, while the second two break them

Distributed Two-Phase Locking

- There are several ways of implementing two-phase locking in distributed systems
- *Centralized 2PL* - a single machine is responsible for granting and releasing locks
- *Primary 2PL* - each data item is assigned a primary copy, and the lock manager on that copy's machine is responsible for granting and releasing locks
- *Distributed 2PL* - schedulers on each machine that has a copy of a data item handle the locking for that machine's copy of the data item

Pessimistic Timestamp Ordering

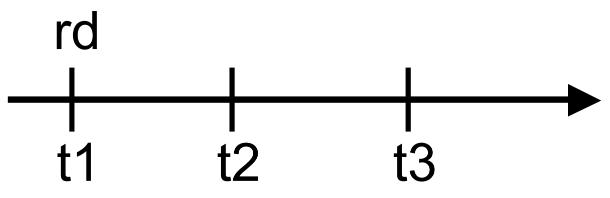
- Every transaction is assigned a timestamp at the moment it starts, and every operation in that transaction carries that timestamp
- Every data item in the system has a read timestamp and a write timestamp, which get updated when a read or write operation occurs to match that operation's timestamp
- If two operations conflict, the data manager processes the one with the lowest timestamp first

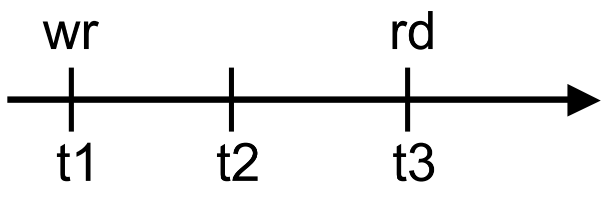
Pessimistic Timestamp Ordering

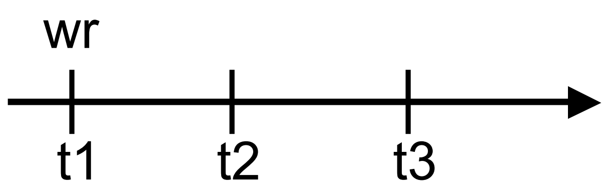
- If a read operation is requested for a piece of data whose write timestamp is later than the reading transaction's timestamp, the reading transaction is aborted
- If a write operation is requested for a piece of data whose read timestamp or write timestamp is later than the writing transaction's timestamp, the writing transaction is aborted
- Can this algorithm deadlock?

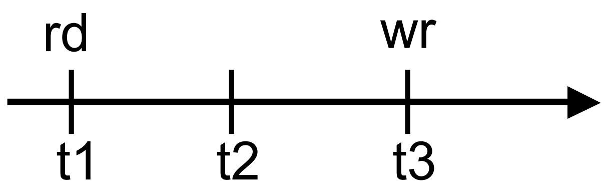
Pessimistic Timestamp Ordering Examples

- Transactions T1, T2, T3, with timestamps t1 < t2 < t3
- Data item x with timestamps rd (read), wr (written), tn (tentatively written)
- What happens when T2 tries to *write* x?



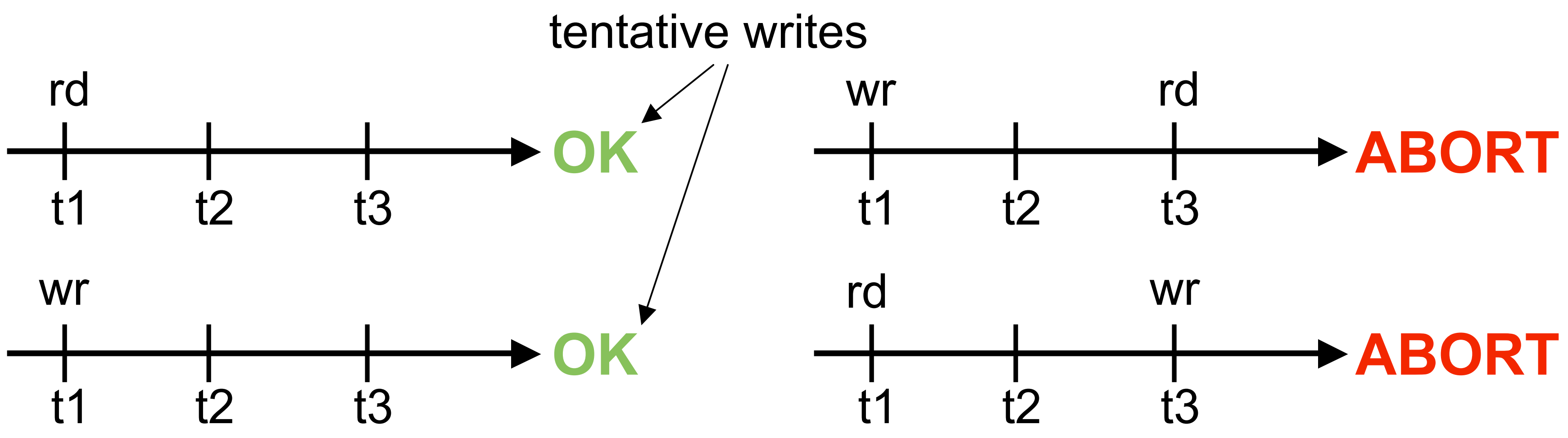






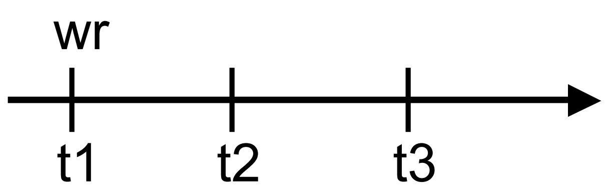
Pessimistic Timestamp Ordering Examples

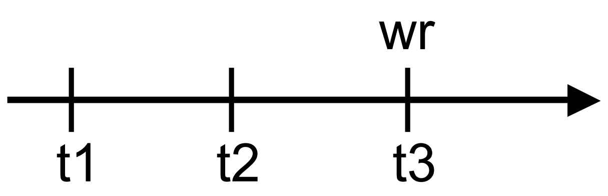
- Transactions T1, T2, T3, with timestamps t1 < t2 < t3
- Data item x with timestamps rd (read), wr (written), tn (tentatively written)
- What happens when T2 tries to *write* x?

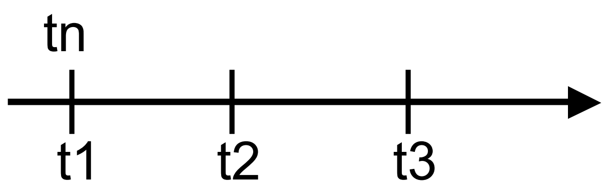


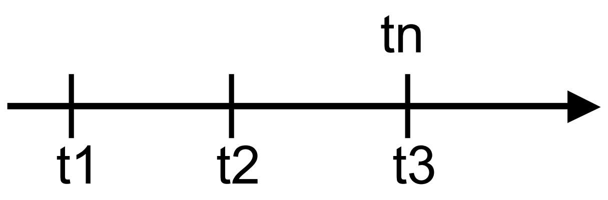
Pessimistic Timestamp Ordering Examples

- Transactions T1, T2, T3, with timestamps t1 < t2 < t3
- Data item x with timestamps rd (read), wr (written), tn (tentatively written)
- What happens when T2 tries to *read* x?



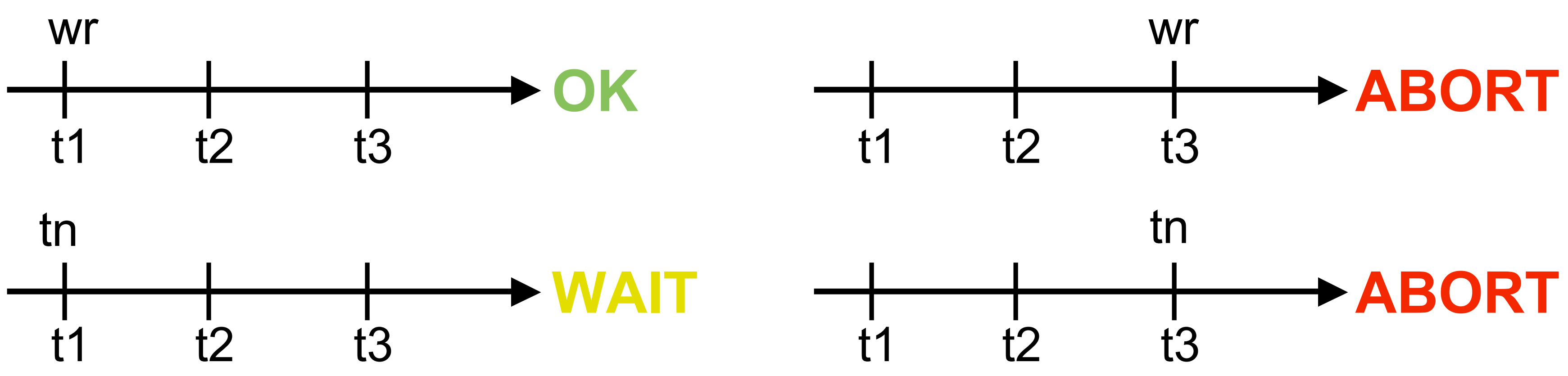






Pessimistic Timestamp Ordering Examples

- Transactions T1, T2, T3, with timestamps t1 < t2 < t3
- Data item x with timestamps rd (read), wr (written), tn (tentatively written)
- What happens when T2 tries to *read* x?



Pessimistic Timestamp Ordering

- If a read operation is requested for a piece of data whose write timestamp is later than the reading transaction's timestamp, the reading transaction is aborted
- If a write operation is requested for a piece of data whose read timestamp or write timestamp is later than the writing transaction's timestamp, the writing transaction is aborted
- Can this algorithm deadlock? *No*

Optimistic Timestamp Ordering

- Execute operations without regard to conflicts, but keep track of timestamps the same way as in pessimistic timestamp ordering
- When the time comes to commit, check to see if any data items used by the transaction have been changed since the transaction started - abort if something has been changed, and commit otherwise
- Can this algorithm deadlock?

Optimistic Timestamp Ordering

- Execute operations without regard to conflicts, but keep track of timestamps the same way as in pessimistic timestamp ordering
- When the time comes to commit, check to see if any data items used by the transaction have been changed since the transaction started - abort if something has been changed, and commit otherwise
- Can this algorithm deadlock? *No*

Optimistic Timestamp Ordering

- Optimistic concurrency control allows maximum parallelism, but at a price…
- If something fails, the effort of an entire transaction has been wasted
- When load increases, conflicts may increase as well, making optimistic concurrency control less desirable

- Not much work has been done on optimistic concurrency control in distributed systems (a potential project area?)

Next Class

- Replication and Consistency Models