CSCE 455/855
Distributed Operating Systems

Transactions

Steve Goddard
goddard@cse.unl.edu

http://www.cse.unl.edu/~goddard/Courses/CSCE855

1

Atomic Transactions

- Transaction
  - » performs a single logical function
  - » all-or-none computation
    - ❖ either all operations are executed or none
  - » must do so in the face of system failures
- Transaction execution
  - » start transaction
  - » series of read and write operations
  - » either a commit or abort operation
    - ❖ commit: all transaction operations executed successfully no transaction operations are allowed to hold
    - ❖ roll back: restore system to original state (before transaction started)

2

Transactions

- Properties of Transactions
  - » atomic: actions occur indivisibly
  - » consistent: system invariants hold
    - ❖ for ex: conservation of money
    - ❖ note that inside transaction this is violated, but from outside, the transaction is indivisible
  - » isolated: transactions do not interfere with each other
    - ❖ aka serializable
    - ❖ looks as though all transactions done in some sequential order
  - » durable: once a transaction commits, results are permanent

3

Example of Serializable Transactions

```
Begin_transaction
    x = 0;
    x = x+1;
End_transaction

Begin_transaction
    x = 0;
    x = x+2;
End_transaction

Begin_transaction
    x = 0;
    x = x+3;
End_transaction
```

4

Transaction Primitives

- Transaction commands
  - » begin-transaction
  - » end-transaction
  - » abort-transaction
    - ❖ must return to state before the begin-transaction
    - ❖ often referred to as “roll-back”
  - » commit-transaction
    - ❖ changes in transaction take effect to outside world
- Transaction operations
  - » read
  - » write
  - » etc...

Transaction Example

- Suppose we have three transactions T1, T2, and T3
  - » two data elements, A and B
  - » scheduled in a round-robing scheduler
  - » one operation per time slice

T1
write(A)
read(A)

T2
read(A)
write(B)
write(A)

T3
write(A)
read(B)

T#	Ts	event1	event2	event3	event4	event5	event6	event7
T1	20	Aw			Ar			
T2	21		Ar			Bw		Aw
T3	22			Aw			Br	

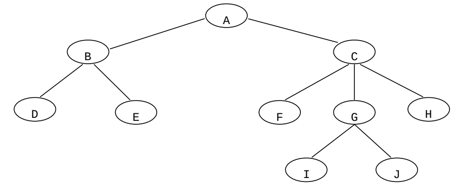
Transaction Example (cont.)

- Objective: find some ordering in which atomicity is preserved
  - » start out T1 → T2 → T3
    - but T1 reads A after T3 writes
    - now we have T3 → T1
    - atomicity is not preserved
    - abort T1
  - » now try T2 → T3 → T1
    - then T2 writes A after T3's write
    - meaning T3 → T2
    - abort T2
  - » now try T3 → T1 → T2
    - this works in the end...

7

Nested Transactions

- Transaction divided into sub-transactions
  - » structured as a hierarchy
  - » internal nodes are masters for its children
  - » advantages:
    - better performance: aborted sub-transactions do not abort masters
    - increased concurrency: only need to lock sub-transactions



8

Nested Transactions (cont.)

- Aborting committed children
  - » suppose a parent transaction starts several child transactions
  - » one or more child commits
    - only after committing is the child's results visible to parent
    - i.e. atomicity is preserved at child level
  - » then parent aborts...
    - but child already "committed"
  - » parent abort must roll back all child transactions
    - even if they have committed

9

Implementing Transactions

- Conceptually, a transaction is given a private workspace
  - » consisting of all resources it has access to
  - » before commit: all operations done to private workspace
  - » after commit: changes are made to actual workspace (file system, etc.)
  - » if the shadowed workspaces of more than one transaction intersects
    - and one of them has a write operation
    - then there is a conflict
    - one of the transactions must be aborted

10

Implementing Transactions (cont.)

- Shadow blocks
  - problem: copying files to a private workspace is expensive!
    - so just copy the blocks that the transaction needs
    - copy index block for file instead of file
  - don't need to copy blocks that are only read
  - demand-driven copying: only copy when a block is first modified
    - a kind of caching
  - write "shadowed" blocks on commit

11

Implementing Transactions Writeahead Log

- Log consists of:
  - transaction name
  - data item name
  - old value
  - new value
- Write log before performing write operations
  - onto non-volatile storage
- Transaction log consists of:
  - <Ti start>
  - series of (Ti, x, old value, new value)
  - <Ti commits> or <Ti aborts>
- Recovery procedures
  - undo(Ti): restores a values written by Ti to old values
  - redo(Ti): sets all values written by Ti to new values

12

Implementing Transactions
Writeahead Log (cont.)

- If Ti aborts:
  - » execute undo(Ti)
- If there is a system failure
  - » can use redo(Ti) to make sure all updates are in place
    - ❖ compare writeahead to actual value
    - ❖ also use the log to proceed with the transaction
  - » if an abort is necessary, use undo(Ti)
- Note that the ‘commit’ operation must be done atomically
  - » difficult when different machines, processes are involved

13

Implementing Transactions
Two-Phase Commit

- Coordinator is selected (transaction initiator)
  - » Phase 1
    - ❖ coordinator writes ‘prepare’ in log
    - ❖ sends ‘prepare’ message to all processes involved in the commit (subordinates)
    - ❖ subordinates write ‘ready’ (or ‘abort’) into log
    - ❖ subordinates reply to coordinator
  - » Phase 2
    - ❖ coordinator logs received replies (or aborts)
    - ❖ coordinator logs ‘commit’ and sends ‘commit’ message
    - ❖ subordinates write ‘commit’ into their log
    - ❖ do the commit
    - ❖ send ‘finished’ message to coordinator

14

Implementing Transactions
Two-phase commit (cont.)

- » If any subordinate cannot commit, abort transaction
  - ❖ if, for example, the subordinate does not respond
- » If all respond, ‘commit’ message makes transaction results stick
  - ❖ i.e. now they are permanent
  - ❖ can remove all transaction log entries, if desired

- ◆ Error recovery in two-phase commit uses log entries
  - » determine when crash occurred
  - » proceed from there
  - » may need to repeat some messages

15

Concurrency Control

- ◆ Transactions may need to run simultaneously
  - » transactions can conflict: one may write to a data item others want to read or write
  - » need methods to synchronize concurrent access
- ◆ Concurrency control methods
  - » locking
  - » optimistic concurrency control
  - » timestamps

16

Locking

- Locks
  - » a semaphore of sorts
  - » read locks: allow *n* read locks on a resource
  - » write locks: no other lock is permitted
- Two-Phase locking
  - » fine-grained locking can lead to deadlock
  - » divide lock requests into two phases
    - growing phase: transaction obtains locks, may not release any
    - shrinking phase: once a lock is released, no locks can be obtained for rest of the transaction

17

Locking

- Disadvantage of two-phase locking
  - » concurrency is reduced
  - » Deadlocks can occur in two-phase locking
    - resource ordering, etc. necessary to prevent deadlocks

18

Two-Phase Locking

- Scenario 1

```
    P1                          P2
lock R1                     lock R1
...                         lock R2
lock R2                     ...
...                         unlock R1
unlock R1                   unlock R2
unlock R2
```

- Scenario 2

```
  P1                           P2
lock R1                      lock R2
...                          lock R1
lock R2                      ...
...                          unlock R1
unlock R1                    unlock R2
unlock R2
```

19

Optimistic Concurrency Control

- Conflicting transactions are rare
  - » therefore let a transaction make all changes
    - ❖ without checking for conflicts
  - » at commit time, check for files that have changed since the transaction began
    - ❖ if so, abort
  - » works best with shadowed implementations
    - ❖ initial changes made to private workspace
  - » distributed transactions need some form of global time
    - ❖ for comparing time for file changes
- Parallelism is maximized
  - » no waiting on locks
  - » inefficient when an abort is needed
  - » not a good strategy in systems with many potential conflicts

20

Timestamp Ordering

- Each transaction assigned a unique timestamp TS(Ti)
  - » if Ti enters system before Tj,
  - » TS(Ti) < TS(Tj)
- Each *data item*, Q, gets two timestamps:
  - » W-timestamp(Q): largest write timestamp
  - » R-timestamp(Q): largest read timestamp
- General concept
  - » process transactions in a serial order
  - » can use the same file, but must do it in order
  - » therefore atomicity is preserved

21

Timestamp Ordering (cont.)

- For a read:

```
if (TS(Ti) < W-timestamp(Q))
{  reject read
   roll back and re-start Ti
}
else /*  TS(Ti) ≥ W-timestamp(Q)  */
{  execute read
   R-timestamp = max(R-timestamp, TS(Ti))
}
```

- Timestamp ordering is deadlock-free
  - » essentially ordering the sequence of file accesses
  - » no cycles can result

22

Timestamp Ordering Example

- Three transactions T1, T2, and T3
  - » two data elements, A and B
  - » scheduled in a round-robing scheduler
  - » one operation per time slice
  - » use read and write timestamps

T1
write(A)
read(A)

T2
read(A)
write(B)
write(A)

T3
write(A)
read(B)

T#	Ts	event1	event2	event3	event4	event5	event6	event7
T1	20							
T2	21							
T3	22							

23

Timestamp Ordering Example

- Three transactions T1, T2, and T3

T1
write(A)
read(A)

T2
read(A)
write(B)
write(A)

T3
write(A)
read(B)

T#	Ts	event1	event2	event3	event4	event5	event6	event7
T1	20	Aw			Ar			
T2	21		Ar			Bw		Aw
T3	22			Aw			Br	

Aw	Ar
10	8
20	21
22	21
21	

Bw	Br
14	16
21	22

24