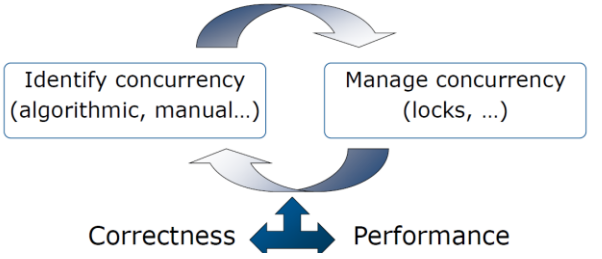
Distributed Transactions

Instructor: Sandhya Dwarkadas
University of Rochester

1

Software Development

Difficulty of Software Development



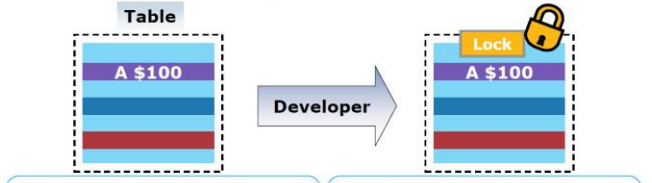
https://software.intel.com/content/dam/develop/external/us/en/documents/sf12-arcs004-100-393551.pdf

2

https://software.intel.com/content/dam/develop/external/us/en/documents/sf12-arcs004-100-393551.pdf

Software Development

The Need for Synchronization



Alice wants $50 from A
- A was $100, A is now $50

Bob wants $60 from A
- A was $100, A is now $40

A should be -10

Alice wants $50 from A
- Alice locks table
- A was $100, A is now $50

Bob wants $60 from A
- Bob waits till lock release
- A was $50, Insufficient funds

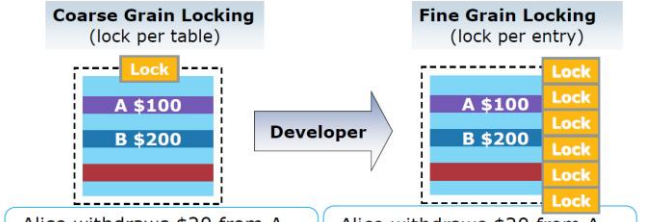
Bob and Alice saw A as $100. Locks prevent such data races

3

https://software.intel.com/content/dam/develop/external/us/en/documents/sf12-arcs004-100-393551.pdf

Software Development

Lock Granularity Optimization



Alice withdraws $20 from A
- Alice locks table

Bob wants $30 from B
- Waits for Alice to free table

Alice withdraws $20 from A
- Alice locks A

Bob wants $30 from B
- Bob locks B

Such Tuning is Time Consuming and Error Prone

4

https://software.intel.com/content/dam/develop/external/us/en/documents/sf12-arcs004-100-393551.pdf

Software Development

Complexity of Fine Grain Locking

Alice transfers $20 from A to B
- Alice locks A and locks B

Performs transfer
- Alice unlocks A and unlocks B

Alice transfers $20 from A to B	Bob transfers $50 from B to A
Locks A	Locks B
Cannot lock B	Cannot lock A

Expensive and Difficult to Debug Millions of Lines of Code

Want coarse grain locking effort for fine grain locking performance

5

Practical Problem I

Shared Array: a[]

Thread 1:	Thread 2:
Write to a[]	Write to a[]

Solution:
1 Spin lock
….

Coarsen-grained? Less lock ops, poor scalability, poor bandwidth.
Fine-grained? Better scalability, more lock cost, more bandwidth

6

Practical Problem II

Shared Array: a[]

Thread 1:	Thread 2:
Read from a[] //parallel	Read from a[] //parallel
Write to a[] // serialized	Write to a[] // serialized

Solution:
1 R/W lock
2 Spin lock or mutex (worse in this case)
….

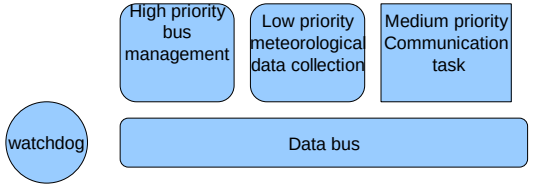
7

Priority inversion

- Occurs when a lower priority process is preempted while holding a lock needed by a high priority process

8

Bad priority inversion: Mars Pathfinder



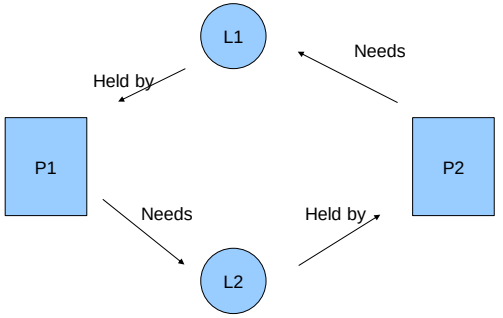
9

Convoying

- Situation where the processes wait in line for the process ahead in the line to finish some task

10

deadlock



11

The Complexity of Locking

– Deadlocks
– Priority Inversion
– Convoy Effect
– Composition and modularity

12

Database Transactions

- Modify multiple data items potentially at multiple locations/by multiple processes as a single atomic operation
- Transaction properties (ACID) –
  - Atomic – happens indivisibly to the outside world
  - Consistent – does not violate system invariants – must hold before and after but not necessarily during
  - Isolated (or serializable) – refers to multiple simultaneous transactions – the final result must appear as if each transaction occurred in some sequential order
  - Durable – once committed, the results become permanent – no failure can undo the results

13

Classification of Transactions

- Flat – series of operations satisfying ACID properties
- Nested – transaction logically divided into sub-transactions
  - Open vs. closed
- Distributed – data is distributed (transaction could be flat)

14

Transaction Implementation

- Private workspace
  - Operations performed on private copy of all open files
- Writeahead log
  - Modify in place but write a log of transaction (id, old, and new values) BEFORE doing so

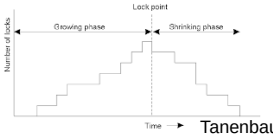
15

Concurrency Control

- Synchronize conflicting read and write operations to ensure serializability
  - Two-phase locking
  - Timestamp ordering
    - Pessimistic vs. optimistic

16

Two-Phase Locking



All schedules of interleaved transactions can be proven to be serializable

Tanenbaum and van Steen: Figure 5-26

- Strict two-phase locking
  - Release all locks at the same time
  - Avoids cascaded aborts
- Problem: deadlocks
  - Solution? Deadlock detection or canonical ordering

17

Timestamp Ordering

- Assign each transaction a unique timestamp (Lamport's)
- Each data item has a (most recent) read and a (most recent) write timestamp
- Lowest timestamp processed first
- Pessimistic timestamp ordering
  - Abort on a conflict as reads and writes occur
- Optimistic timestamp ordering
  - Delay check until time of commit (best with private workspaces)

18

Pessimistic Timestamp Ordering

- Read(T,x)
  - Tts < $ts_{WR}(x)$ → abort
  - Tts > $ts_{WR}(x)$ → perform
  - $ts_{RD}(x) = \max\{Tts, ts_{RD}(x)\}$
- Write(T,x)
  - Tts < $ts_{RD}(x)$ → abort
  - Tts > $ts_{RD}(x)$, $ts_{WR}(x)$ → perform
  - $ts_{WR}(x) = \max\{Tts, ts_{WR}(x)\}$

19

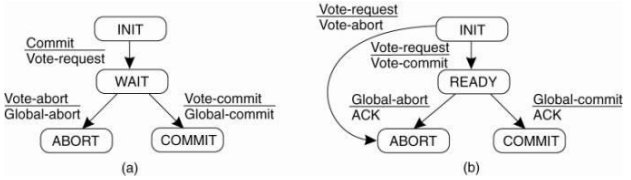
Distributed Commit

Operation must be performed by each member of a process group or none at all

- Established by means of a coordinator
- 1-Phase commit?
  - No way to tell the coordinator that the operation cannot be performed
- 2-Phase commit
- 3-Phase commit

20

Two-Phase Commit



Tanenbaum and van Steen Figure 7.17:
Finite state machines for coordinator (a) and participant (b)

21

Two-Phase Commit

- Phase 1
  - Step 1: vote request
  - Step 2 – return vote commit or abort
- Phase 2 – global commit or abort

Problem: failures when blocked waiting for incoming messages

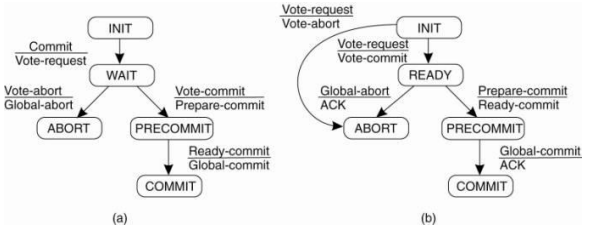
- (a) Participant in INIT state; (b) coordinator in WAIT state; (c) participant in READY state

Solutions: timeout; ABORT under (a) and (b), poll other participants under (c)

Remaining problem: must wait for coordinator under (c) if all participants in READY state

22

Three-Phase Commit



Tanenbaum and van Steen Figure 7-21:
Finite state machines for coordinator (a) and participant (b)

23

Three-Phase Commit

- No state from which it is possible to transition directly to either COMMIT or ABORT
- No state from which a final decision cannot be made on failure and from which a transition to COMMIT is possible
- No crashed process could be in COMMIT if any participant is in READY state; or in INIT or ABORT if any participant is in PRE-COMMIT
  - Allows a participant to use a majority to decide whether to ABORT (if majority is in READY) or COMMIT (if majority is in PRE-COMMIT) when coordinator is unresponsive

24