COMP90020: Distributed Algorithms

10. Algorithms for Concurrency Control

Lock and Wait

Miquel Ramirez

THE UNIVERSITY OF MELBOURNE

Semester 1, 2019

Agenda

Agenda

1. Locks
2. Queue Locks
3. Deadlock Detection
4. Discussion
5. Biblio & Further Reading

Locking and Mutual Exclusion: Basic Idea

Goal: Consistency & Isolation – the "CI" in ACID

Concurrent transactions need to be *scheduled* so effects on shared objects are serially equivalent.

Strategy: *serialize* access to shared objects "locking" them

- Require process p to lock an object o, and release o when done
- A shared object o can only be read or written to by a single process p at a time
- Processes p' interested on o will have to wait for the lock to be released.

Mutual Exclusion algorithms and data structures implement serialization.

Review: Serial Equivalence and Scheduling

Serializing Transactions = Constrain Schedule of Operations Over Time

Let s and t be *transactions*, no a priori order between them,

- if there are no conflicting operations, these can be reordered arbitrarily e.g. s then t, t then s or any interleaving of operations,
- otherwise all pairs of conflicting operations must be executed in the same order on all objects being accessed concurrently and for every possible schedule.

Conflicting Operation

Let s and t be *transactions* with ops $\alpha(o) \in s$ and $\beta(o) \in t$ over objects o, let $\mathcal{R}(o)$ and $\mathcal{W}(o)$ be sets of ops that respectively read and write the value of an object o. Then for every pair $\langle \alpha(o), \beta(o) \rangle$

- if both in $\mathcal{R}$, then operations do not conflict,
- when at least one in $\mathcal{W}$ then operations are conflicting.

Locks - Rules

Serialization can be achieved with locks on shared objects.

→ Locks must be obtained in line with desired serialization order.

Locks - Rules

Serialization can be achieved with locks on shared objects.

→ Locks must be obtained in line with desired serialization order.

Transactions can hold two types of locks: read or write:

- Multiple transactions may concurrently get the same read lock.
- While a write lock is held, no other transaction holds this write or the corresponding read lock.
- Read locks can only be promoted to write locks when all other read locks on same shared object released,
- once a transaction releases a lock, it cannot claim any others.

Question: That Last Restriction...

Question: That Last Restriction...

Transactions t and s modify Account shared objects a and b, initially a and b equal to 25, goal is that both a and b equal to 250

Transaction t		Transaction s	
i	Operation	i	Operation
1	$lock(a)$, $y \leftarrow a.getBalance()$	1	$lock(a)$, $z \leftarrow a.getBalance()$
2	$y \leftarrow y + 100$	2	$z \leftarrow 2\,z$
3	$a.deposit(y)$, $unlock(a)$	3	$a.deposit(z)$, $unlock(a)$
4	$lock(b)$, $y \leftarrow b.getBalance()$	4	$lock(b)$, $z \leftarrow b.getBalance()$
5	$y \leftarrow y + 100$	5	$z \leftarrow 2\,z$
6	$b.deposit(y)$, $unlock(b)$	6	$b.deposit(z)$, $unlock(b)$

Question!

Why cannot a transaction claim any locks after releasing one?

(A): Serialization not guaranteed

(B): That is a typo in your slides...

Question: That Last Restriction...

Transactions t and s modify Account shared objects a and b, initially a and b equal to 25, goal is that both a and b equal to 250

Transaction t		Transaction s	
i	Operation	i	Operation
1	$lock(a)$, $y \leftarrow a.getBalance()$	1	$lock(a)$, $z \leftarrow a.getBalance()$
2	$y \leftarrow y + 100$	2	$z \leftarrow 2\,z$
3	$a.deposit(y)$, $unlock(a)$	3	$a.deposit(z)$, $unlock(a)$
4	$lock(b)$, $y \leftarrow b.getBalance()$	4	$lock(b)$, $z \leftarrow b.getBalance()$
5	$y \leftarrow y + 100$	5	$z \leftarrow 2\,z$
6	$b.deposit(y)$, $unlock(b)$	6	$b.deposit(z)$, $unlock(b)$

Question!

Why cannot a transaction claim any locks after releasing one?

(A): Serialization not guaranteed

(B): That is a typo in your slides...

$\rightarrow$ (A): Serialization cannot guaranteed, see next slide

If You Lock after an Unlock...

	Transaction t	Transaction s	Shared	
$Time$	Operation	Operation	a	b
1	$lock(a),\ y \leftarrow a.getBalance()$	–	25	25
2	$y \leftarrow y + 100$	–		
3	$a.deposit(y),\ unlock(a)$	–	125	
4	–	$lock(a),\ z \leftarrow a.getBalance()$		
5	–	$z \leftarrow 2\,z$		
6	–	$a.deposit(z),\ unlock(a)$	250	
7	–	$lock(b),\ z \leftarrow b.getBalance()$		
8	–	$z \leftarrow 2\,z$		
9	–	$b.deposit(z),\ unlock(b)$		50
10	$lock(b),\ y \leftarrow b.getBalance()$	–		
11	$y \leftarrow y + 100$	–		
12	$b.deposit(y),\ unlock(b)$	–		150

Serial equivalence does not hold, this execution ends in different state.

Two-phase locking

Two-phase locking consists of two subsequent phases (per transaction):

- a growing phase, in which locks are accumulated; and
- a shrinking phase, in which the acquired locks are released.

Two-phase locking

Two-phase locking consists of two subsequent phases (per transaction):

- a growing phase, in which locks are accumulated; and
- a shrinking phase, in which the acquired locks are released.

More Rules

- Read locks can be released early on in the shrinking phase.
- Write locks can only be released after committing or aborting
- Shared objects have the written value in case of a commit, or the original value in case of an abort.

Two-phase locking - Example

Goal

t wants to add 10 dollars and s wants to add 20 dollars to a.

Prudent lock management disallows t and s from concurrently obtain the read lock of a.

Two-phase locking - Example

Goal

t wants to add 10 dollars and s wants to add 20 dollars to a.

Prudent lock management disallows t and s from concurrently obtain the read lock of a.

	Transaction t	Transaction s	Shared
$Time$	Operation	Operation	a
1	$lock(a)$, $a.deposit(10)$	–	
2	$commit$, $unlock(a)$	–	+10
3	–	$lock(a)$, $a.deposit(20)$	
4	–	$commit$, $unlock(a)$	+30

Result: value of a increased by 30 dollars by adding 10 and 20 dollars in some sequential order.

Mutual Exclusion Algorithms

Problem: multiple process p_i want access to shared object o

- *Locking* privileges one single process p, which is granted access,
- privileged process are said to *enter* critical section,
- which is *left* when process releases the lock.

Mutual exclusion DA's need to satisfy following conditions in every execution h

- Mutual exclusion: in every configuration γ, at most one process p_i is privileged.
- Starvation–freeness: when process p_i tries to enter *critical section*, and no process p_j is privileged forever, then p_i will be eventually be privileged.

Example: Transactions with Exclusive Locks

Consider transactions over shared objects a, b, c, all start unlocked

Transaction t		Transaction s	
i	Operation	i	Operation
1	$x \leftarrow b.getBalance()$	1	$x \leftarrow b.getBalance()$
2	$b.setBalance(1.1x)$	2	$b.setBalance(1.1x)$
3	$a.withdraw(0.1x)$	3	$c.withdraw(0.1x)$

Question!

What pairs of operations are conflicting?

(A): (t_1, s_1)

(B): (t_2, s_2)

(C): $(t_2, s_3), (t_3, s_2)$

(D): $(t_1, s_2), (t_2, s_1)$

Example: Transactions with Exclusive Locks

Consider transactions over shared objects a, b, c, all start unlocked

Transaction t		Transaction s	
i	Operation	i	Operation
1	$x \leftarrow b.getBalance()$	1	$x \leftarrow b.getBalance()$
2	$b.setBalance(1.1x)$	2	$b.setBalance(1.1x)$
3	$a.withdraw(0.1x)$	3	$c.withdraw(0.1x)$

Question!

What pairs of operations are conflicting?

(A): (t_1, s_1)

(B): (t_2, s_2)

(C): $(t_2, s_3), (t_3, s_2)$

(D): $(t_1, s_2), (t_2, s_1)$

$\rightarrow$ (B & D): are conflicting the process running s and t may read and write concurrently to object b

Example: Serialization via Locking

Transaction t			Transaction s		
Time	Operation	Locks	Time	Operation	Locks
1	$start$	–	1		
2	$x \leftarrow b.getBalance()$	b	2	$start$	
3	$b.setBalance(1.1x)$	b	3	$x \leftarrow b.getBalance()$	wait b
4	$a.withdraw(0.1)$	a, b	4	–	wait b
5	$commit$	–	5	–	wait b
6		–	6	$x \leftarrow b.getBalance()$	b
7		–	7	$b.setBalance(1.1x)$	b
7		–	7	$c.withdraw(0.1x)$	b, c
8		–	8	$commit$	–

Don't Start Starve

Transaction t			Transaction s		
Time	Operation	Locks	Time	Operation	Locks
1	$start$	–	1		
2	$x \leftarrow b.getBalance()$	b	2	$start$	
3	$b.setBalance(1.1x)$	b	3	$x \leftarrow b.getBalance()$	wait b
4	$a.withdraw(0.1)$	a, b	4	–	wait b
5	$commit$	–	5	–	wait b
6		–	6	$x \leftarrow b.getBalance()$	b
7		–	7	$b.setBalance(1.1x)$	b
7		–	7	$c.withdraw(0.1x)$	b, c
8		–	8	$commit$	–

Question!

The person programming t forgot to initialize a, and the process crashes. What happens to s?

(A): s keeps waiting forever

(B): s becomes privileged

(C): s aborts

(D): s loses the update from t

Don't Start Starve

Transaction t			Transaction s		
Time	Operation	Locks	Time	Operation	Locks
1	$start$	–	1		
2	$x \leftarrow b.getBalance()$	b	2	$start$	
3	$b.setBalance(1.1x)$	b	3	$x \leftarrow b.getBalance()$	wait b
4	$a.withdraw(0.1)$	a, b	4	–	wait b
5	$commit$	–	5	–	wait b
6		–	6	$x \leftarrow b.getBalance()$	b
7		–	7	$b.setBalance(1.1x)$	b
7		–	7	$c.withdraw(0.1x)$	b, c
8		–	8	$commit$	–

Question!

The person programming t forgot to initialize a, and the process crashes. What happens to s?

(A): s keeps waiting forever

(B): s becomes privileged

(C): s aborts

(D): s loses the update from t

$\rightarrow$ (A): in the absence of any protocol to manage mutual exclusion, s waits forever

Agenda

1. Locks
2. Queue Locks
3. Deadlock Detection
4. Discussion
5. Biblio & Further Reading

Basic Locks: Test-And-Set Lock

Idea

all process p_i access *Boolean* shared object l supporting test-and-set

→ l can be used to control access over many objects o,

→ Initialization: l starts being $false$

In order to become privileged p_i executes

Test-And-Set Lock

1. $b \leftarrow$ test-and-set(l)
2. if b is $false$ goto 1.

- Simple conceptually, but poor performance

Basic Locks: Test-And-Set Lock

Idea

all process p_i access *Boolean* shared object l supporting test-and-set

→ l can be used to control access over many objects o,

→ Initialization: l starts being $false$

In order to become privileged p_i executes

Test-And-Set Lock

1. $b \leftarrow$ test-and-set(l)
2. if b is $false$ goto 1.

- Simple conceptually, but poor performance

Lots of failed attempts, high congestion

Idea: Spinning on Local Copies

To read from shared object in a loop until value changes.

Test-And-Test-And-Set Lock with Exponential Back-Off

Idea: all process p_i access *Boolean* shared object l supporting test-and-set

$\rightarrow$ Initialization: l starts being $false$, Release: set l to $false$

In order to become privileged p_i executes

Test-And-Test-And-Set Lock

1. $t \leftarrow$ random number
2. $b \leftarrow$ read(l)
3. if b is $true$ then goto 2.
4. $b \leftarrow$ test-and-set(l)
5. if b is $false$ then return
6. sleep t, $t \leftarrow kt$
7. goto 2.

Test-And-Test-And-Set Lock with Exponential Back-Off

Idea: all process p_i access *Boolean* shared object l supporting test-and-set

→ Initialization: l starts being $false$, Release: set l to $false$

In order to become privileged p_i executes

Test-And-Test-And-Set Lock

1. $t \leftarrow$ random number
2. $b \leftarrow$ read(l)
3. if b is $true$ then goto 2.
4. $b \leftarrow$ test-and-set(l)
5. if b is $false$ then return
6. sleep t, $t \leftarrow kt$
7. goto 2.

Pros:

- Easy to implement,
- very efficient if contention is low.
- Avoids thundering herd
  Problem: all waiting process call test-and-set simultaneously

Test-And-Test-And-Set Lock with Exponential Back-Off

Idea: all process p_i access *Boolean* shared object l supporting test-and-set

→ Initialization: l starts being $false$, Release: set l to $false$

In order to become privileged p_i executes

Test-And-Test-And-Set Lock

1. $t \leftarrow$ random number
2. $b \leftarrow$ read(l)
3. if b is $true$ then goto 2.
4. $b \leftarrow$ test-and-set(l)
5. if b is $false$ then return
6. sleep t, $t \leftarrow kt$
7. goto 2.

Pros:

- Easy to implement,
- very efficient if contention is low.
- Avoids thundering herd
  Problem: all waiting process call test-and-set simultaneously

Cons: starvation due to pessimistic k, spinning on l causes congestion

CLH Lock

Queue Locks Idea

Waiting process queued, First In, holds lock, spin on different shared objects (one per waiting process).

CLH Lock

Queue Locks Idea

Waiting process queued, First In, holds lock, spin on different shared objects (one per waiting process).

CLH lock:

- Queue implemented as a dynamic list
- Proc p adds shared object e to queue, set e to $true$, get pointer to previous elem e'
- p spins on previous element e', until $false$
- set e to $true$, do work, release by setting e to $false$,
- once lock *released*, p removes itself from the queue.

CLH Locks with Timeouts

Waiting p wants to give up? →Needs give way to successor in list

Shared objects e for each waiting process p need to have several values:

- $\perp$ if p is waiting or is in the critical section,
- pointer to predecessors' e, if p has given up,
- pointer to special element *released*, if p has left crit section.

Protocols

To acquire lock:

- if queue empty, lock is acquired, otherwise get e for last process in queue, spin until pointer to released read

To give up waiting

- check if p last in queue, if not p updates e to the value of its predecessor.

To release lock

- When p releasing check if p last, otherwise set e to *released*

Quiz: Queue Locks and Byzantine Failures

Question!

Consider the case of a CLH lock where we have three processesses, p_1, p_2, and p_3 waiting on the queue. After a while, p_2 becomes Byzantine. What is the worst possible effect of p_2 becoming Byzantine?

- (A): Comms slowdown due to spinning
- (B): p_2 gets kicked out by p_3
- (C): p_3 gets held out indefinitely
- (D): p_2 leaves the queue

Quiz: Queue Locks and Byzantine Failures

Question!

Consider the case of a CLH lock where we have three processesses, p_1, p_2, and p_3 waiting on the queue. After a while, p_2 becomes Byzantine. What is the worst possible effect of p_2 becoming Byzantine?

(A): Comms slowdown due to spinning

(B): p_2 gets kicked out by p_3

(C): p_3 gets held out indefinitely

(D): p_2 leaves the queue

→ (A & C): in the absence of a third party keeping track of how long processesses have been in the queue it is possible that p_2 freezes the queue if it keeps consistenly and indefinitely sending a message to acquire the lock. This behaviour, which is consistent with the Byzantine failure model, also will congest the communications channel between the queue and other process.

Agenda

Deadlocks

Crucial problem in DS, all process waiting for something to happen

Communication Deadlock

Every process is waiting for some other process to send message to group

Resource Deadlock

Every process waiting for some other process to release lock on shared object

Configuration of DS constantly monitored for cyclic dependencies

- Take snapshot, look for cycles,
- If cycles found, reset DS and rollback to previously taken snapshot.

Cyclic dependency detection enabled by keeping a Wait-for graph, depicting dependencies between *processesses* and *shared objects*.

Wait-For Graphs

Graph $W = (V, E)$ where

- V is set of ongoing transactions, t, s, etc.
- $E = (s, o, t)$ where s and t and transaction, o is shared object with conflicting ops

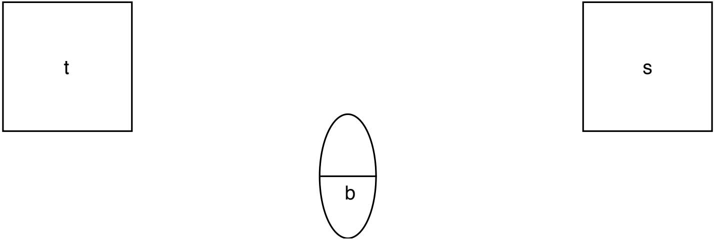
W is managed by lock manager

- Keeps track of which nodes in V are blocked or non-blocked,
- an edge is added whenever s attempts to lock o, already locked by t,
- an edge is removed whenever t releases locks on o,
- nodes are non-blocked if they do not have any outogoing edges.

Example: Transactions with Exclusive Locks

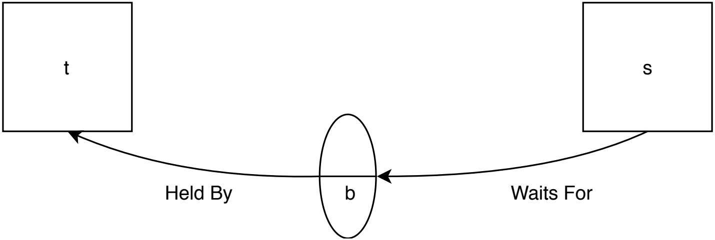
Transactions t, s shared objects a, b, c, all start unlocked

Transaction t		Transaction s	
i	Operation	i	Operation
1	$x \leftarrow b.getBalance()$	1	$x \leftarrow b.getBalance()$
2	$b.setBalance(1.1x)$	2	$b.setBalance(1.1x)$
3	$a.withdraw(0.1x)$	3	$c.withdraw(0.1x)$



Example: s waits for t to finish with b

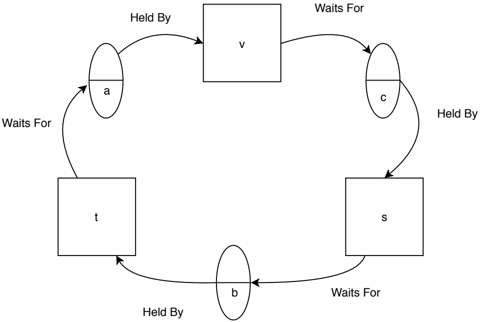
Transaction t			Transaction s		
Time	Operation	Locks	Time	Operation	Locks
1	$start$	–	1		
2	$x \leftarrow b.getBalance()$	b	2	$start$	
3	$b.setBalance(1.1x)$	b	3	$x \leftarrow b.getBalance()$	wait b



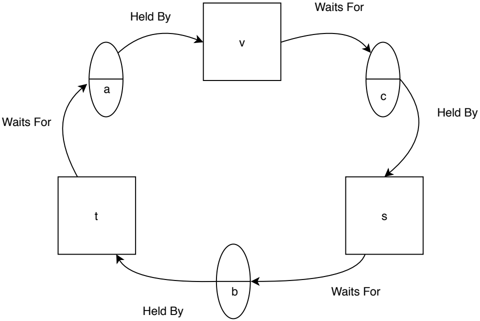
In English: "s waits for b to be available, b lock is held by t"

Cycles in Wait-For Graphs

Transaction t			Transaction s		
Time	Operation	Locks	Time	Operation	Locks
1	$start$	–	1		
2	$x \leftarrow b.getBalance()$	b	2	$start$	
3	$b.setBalance(1.1x)$	b	3	$x \leftarrow b.getBalance()$	wait b, c
4	$a.withdraw(0.1x)$	wait a,b	4	–	wait b



Cycles in Wait-For Graphs



Cycle $t \xrightarrow{a} v \xrightarrow{c} s \xrightarrow{b} t$

- All transactions blocked waiting for locks
- No lock can be released (deadlock)
- If one transaction is *aborted*, then locks are released and that cycle is broken

Lazy Deadlock Avoidance with Lock Timeouts

Idea for Lock Timeouts

Add a timer and a Boolean flag $v(l)$ that indicates whether lock l is vulnerable or not.

1. When timer reaches deadline set
   - $v(l) \leftarrow true$
2. If $v(l)$ is $true$ and no transaction waiting on it
   - nothing happens,
3. otherwise l is released
   - transaction holding lock is aborted to maintain ACID

Limitations of Lock Timeouts

- Locks may be broken by a waiting transaction when there is no deadlock
- If DS is overloaded,
  - lock timeouts will happen more often
  - long, CPU intensive transactions will be penalised
- Difficult to select a suitable length for deadline

Active Deadlock Detection

Idea

Deadlock detection part of the *Coordinator/Lock Manager*, that holds a unique, centralized instance of the wait-for graph W.

Manager checks W for cycles regularly

- Edges are added and removed from W by the lock manager's interface to acquire and relase locks.
- Challenge #1: efficient cycle detection
  - R. E. Tarjan, "Depth-First Search and Linear Graph Algorithms" (1972)
- Challenge #2: When cycle detected, choose a transaction to be aborted
  - longest running, or the one in the most cycles

Locking is Pessimistic

Question!

Two processesses are concurrently incrementing the values of n shared objects, starting a transaction to do so that locks the selected object. How often will the lock actually prevent a problem (on average)?

(A): for every transaction

(B): once every n^2 transactions

(C): every 42 transactions

(D): once every n transactions

Locking is Pessimistic

Question!

Two processesses are concurrently incrementing the values of n shared objects, starting a transaction to do so that locks the selected object. How often will the lock actually prevent a problem (on average)?

(A): for every transaction

(B): once every n^2 transactions

(C): every 42 transactions

(D): once every n transactions

$\rightarrow$ (B): All things being equal, either process is equally likely to initiate a transaction for any of the n shared objects, and do so independently. Hence the probability of the two processesses competing for locking a given shared object is the product of the probabilities of choosing that object: $\frac{1}{n}\frac{1}{n} = \frac{1}{n^2}$.

Agenda

1. Locks
2. Queue Locks
3. Deadlock Detection
4. Discussion
5. Biblio & Further Reading

Drawbacks of Locking

With locking we get the problem of deadlocks

- Preempting and breaking deadlocks reduces concurrency,
- lock timeouts convey wasting CPU time when transactions are aborted preemptively,
- implementation of intelligent lock manager non trivial,
- the manager itself is a single point of failure,
- "Groundhog Day": fixing deadlocks can trigger recurrent cascading aborts .

Locks require disciplined programming

- excessive locking can make deadlock resolution intractable,
- each failed lock acquisition incurs significant comms or processessing overhead.

Agenda

1. Locks
2. Queue Locks
3. Deadlock Detection
4. Discussion
5. Biblio & Further Reading

Further Reading

Coulouris et al. *Distributed Systems: Concepts & Design*

- Chapter 16, Section 16.4, 16.5, 16.6

Fokkink *Distributed Systems: An Intuitive Approach*

- Chapter 5 – Deadlock Detection
- Chapter 14 – Mutual Exclusion

Breitbart et al *"On Rigorous Transaction Scheduling"*

Y. Breitbart, D. Georgakopoulos, M. Rusinkiewicz, A. Silberschatz
IEEE Transactions on Software Engineering (TSE),
September 1991, Volume 17, Issue 9, pp. 954-960