Distributed Database Systems
Scheduling and Deadlock

Instructor: Randal Burns
Lecture for March 11, 2000
Computer Science 600.416
Johns Hopkins University

Scheduling and Deadlock
Overview

• Scheduling
• Priority Inversion
• Convoys
• Deadlock
  – Definitions
  – Detection
  – Avoidance
Scheduling

• The lock manager is a scheduler
  – If many transactions are waiting on a lock, the lock manager
decides who will advance

• Traditionally. LMs uses FIFO scheduling
  – Leads to the convoy phenomena when many transactions are using
  the same resource
  – Poor resource utilization

• Simple fix, add priority to locks
  – Advance higher priority trans. in favor of lower priorities
  – Does not make convoys go away, and introduces the priority
inversion problem
Priority Inversion

• Occurs in two different fashions
  – Low priority tasks get scheduled at high priority
    • *E.g.*, a DB server executes at its (the servers) priority when conducting tasks on behalf of the client
    • Solution, DB server differentiates between its tasks (high priority) and its client’s tasks, which run at the priority of the client
  – Or low priority tasks sleep with high priority holdings
    • MARS pathfinder problem, low priority jobs held a system-wide important resource, in this case a mutex on the data bus
    • All jobs need to wait for low priority task to complete before making progress, but the low priority task gets scheduled infrequently
    • Solution, upgrade low priority task to the level of highest priority task waiting on the resource that it holds
  – Both solutions: *priority inheritance*
Convoy

• Convoy is or forms on a high traffic lock
  – Often the log lock in a database
    • Each update generates an entry at end of log
    • Few hundred instructions
  – Supposed process P1 holds log lock and is preempted by P2, P3, …, Pk who all do an update than wait for log lock
    • All of P2, …, Pk wait for P1
    • When P1 gets scheduled, it does an update and gets in the back of the line
    • These convoys are stable and long lived
  – Priority locks do not solve, as priorities implement a FIFO at each priority level. Get a convoy at each priority and starvation of low priority locks.

Scheduling and Deadlock
Solving Convoys

• 3 part solution
  – Do not schedule hotspot locks on a FIFO basis, rather wake up everybody and let scheduler decide who should run
    • Integrate lock dispatcher with O/S scheduler
    • This is the pthreads approach
  – In a multiprocessor, spin (busy wait) on lock for a few hundred instructions
    • Avoids overheads of callbacks and unschedule/reschedule process
    • Particularly effective when locked resource is not busy
    • Optimistic technique in a pessimistic locking environment
  – Do not allow systems holding hotspot locks to be preempted
    • Again integrate O/S scheduler with locking

Scheduling and Deadlock
What are the obstacles to O/S Integration with LMs?

- Scheduling is done in applications
  - O/S has no knowledge
  - This is the DB problem

- Managed resources are external to the O/S
  - The system bus on the MARS Pathfinder

- *But is this really a problem, because the O/S can own the scheduling primitives?*
  - In some apps no. For example, all MARS needed was a mutex, which the O/S can manage
  - But DBs need a richer set of constructs than the O/S provides

Scheduling and Deadlock
Deadlock

• Deadlock occurs when transactions wait for each other
  – Convoys and priority inversions are merely slowdowns
  – Deadlock is the “ultimate” slowdown
• Deadlocks are permanent situations
• Two approaches to solving deadlock
  – Avoidance – deadlock free locking protocols
  – Detection – allow deadlocks to occur and resolve them through rollback
• These approaches are pessimistic and optimistic
  – Avoidance incurs overhead at all times to avoid deadlock
  – Detection only pays a price when deadlocks occur,
    • There is actually substructure to detection

Scheduling and Deadlock
Conditions for Deadlock

• Deadlock setup
  – Two transactions: T1 and T2
  – Two resources: A and B
  – T1 holds A wants B
  – T2 holds B wants A

• Deadlock conditions
  – Neither transaction is willing to release its current holding
Timeout: Optimistic Detection

- Optimism: deadlocks occur very infrequently and therefore a system should incur no runtime costs
  - Pay a big price when deadlocks occur
- Place a long timeout when waiting on a resource, if the timeout occurs, declare a deadlock and rollback the transaction
- All systems must rely on timeouts at some level
  - Deal with media failures or external failures
- Tunable amount of optimism, set the timeout
- May not actually detect deadlock, may detect long waits
Waits for Detection

• Detect actual deadlocks
• Encode resources dependencies in a waits for graph
  – Figure GR 7.16
  – Any cycle in the waits for graph is a deadlock
• Detects actual deadlocks
• Detects deadlocks when they occur
• Rely on timeout as a last resort
• Less optimistic (incur some runtime cost) more realistic
  (deadlocks occur and we need to find them)
Deadlock Avoidance

• No deadlocks occur if there are no cycles in the waits for graph
  – Order all of the locks in the systems and all transactions acquire locks in this order
  – Transactions are either ahead or behind of other transactions, but no wait cycles occur

• There are also avoidance techniques based on a pre-declaration of all locks that a transaction will use. *Why is this not practical?*
  – Need an on-line, dynamic solution
  – Not a useful interface, few applications know or can express their needs ahead of time