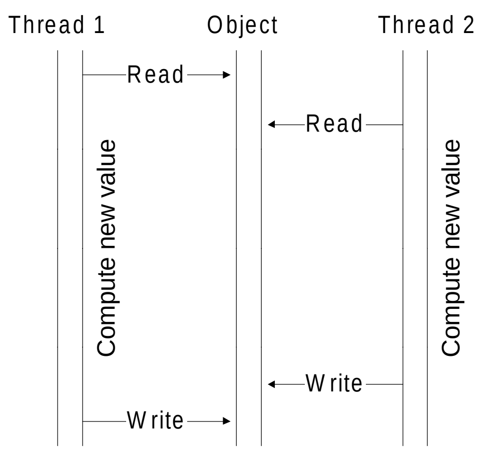
Threaded Applications and Concurrency Control

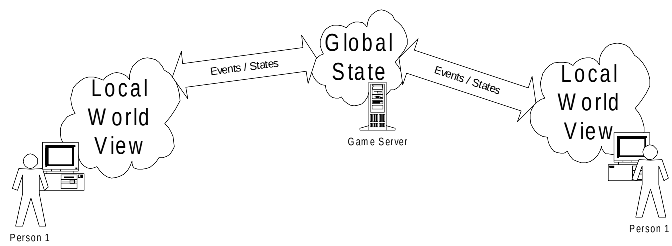
Applicable to multi-threaded, multi-process, and distributed applications

The Lost Update Problem



- Thread 2 update is lost
- Could be
  - Disk TOC
  - Flight reservation
  - Game world
- Makes reasoning hard
  - Hard to think about all possible interleavings of threads

Lost Update - Multi-Player Game



- Consider: Person 1 and Person 2 each take treasure
  - Each locally thinks "I got it before he did"
  - Result is inconsistent local worlds

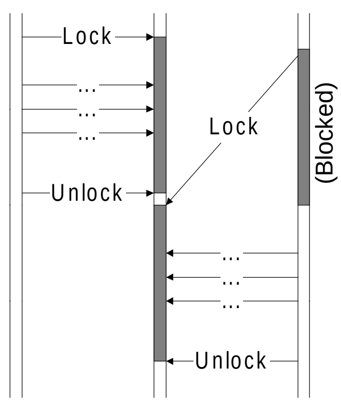
Reasoning Levels

- Individual interleavings
  - Good for informal reasoning and design
  - Too many to enumerate exhaustively
- Finite-state models (state machines)
  - Petri nets, process algebra, …
  - Possible but difficult
    - Use when necessary to design isolated protocols
- Idioms & standard protocols
  - Overall patterns with known properties

Concurrency Control Protocols

- Objective: Pretending atomicity
  - Treat concurrent activities as if they occurred serially
    - So that reasoning about interleavings is not needed
- Transactions = Units of (Pretend) Atomicity
  - As if only complete transactions were interleaved
  - Typically a complete read/compute/write sequence
  - Enough to retain globally consistent state
    - But not too much; atomicity and performance are in tension

Mutual Exclusion (Locking)



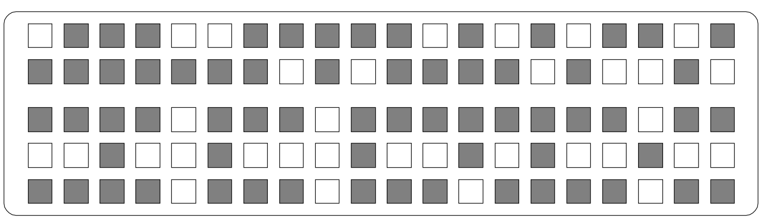
- Basic mechanism
  - Locks or semaphores associated with the shared resource
  - Example: Java “synchronized” classes
- Limitations
  - Atomicity only with respect to the locked resource
    - Not aggregations
  - Performance and responsiveness
    - esp. for aggregations

CREW Locking

- CREW = Concurrent read, exclusive write
  - Reduces blocking when some threads are “pure readers” with respect to globally consistent state
  - *Careful* --- Independent CREW locking does not provide global consistency (see next slide on granularity)

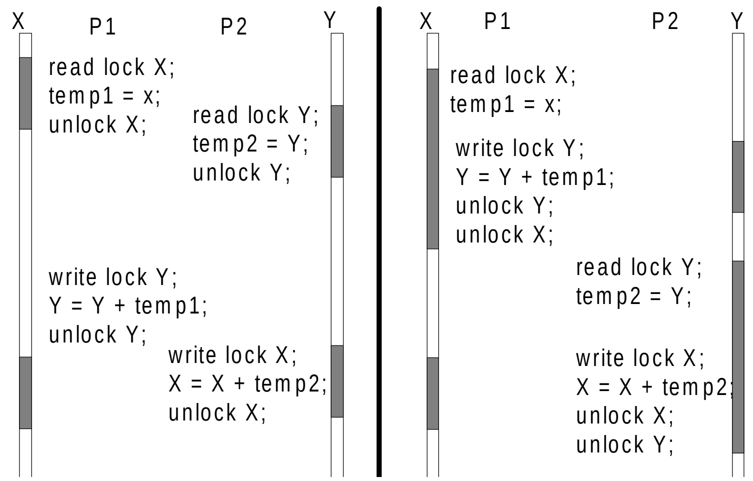
	Locked for Read	Locked for Writing
Obtain Read Lock	OK	*block*
Obtain Write Lock	*block*	*block*

Granularity of Locking



- Airline reservation task:
  - Two seats, together, on EUG->SFO and SFO->LAX
- Locking level
  - The whole airplane, or individual seats?
  - One flight, or all three?
- Coarse grain = easy consistency, lousy performance

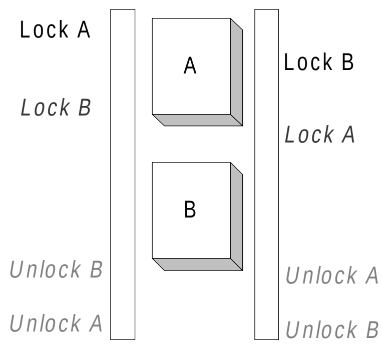
Two-Phase Locking



Two-Phase Locking

- Two-phase locking rule
  - Locking phase: Only lock, no unlocking
  - Unlocking phase: Only unlocking, no more locking
  - Transaction = Locking phase + Unlocking phase
- Theorem
  - Transactions with two-phase locking are serializable
  - Translation: As if the whole collection of items were locked; as if transactions did not overlap

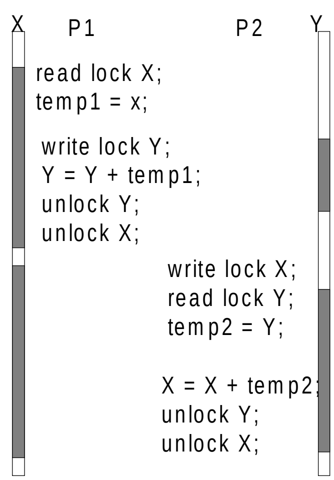
Deadlock & Resource Ordering



- Circular wait
  - First process locks A, waits to lock B
  - Second process locks B, waits to lock A
- Variations
  - any number of resources or locks
- Avoidance
  - Globally consistent order for locking resources

Two-Phase + Resource Ordering

- Pattern of locking/unlocking in two-phase locking with resource ordering is proper nesting of locks used by each transaction
- Natural fit with "monitor" constructs (e.g., Java classes with synchronized methods)



Alternatives to Locking?

- Locking is pessimistic concurrency control
  - Block to prevent inconsistency before it happens
- Alternative: Optimistic concurrency control
  - "Abort" if conflicts cannot be resolved
  - Examples:
    - RCS vs. CVS version management system
      - RCS locks to prevent conflict, CVS allows parallel editing but may not be able to commit all changes
    - Airline reservations
- Variations
  - Tree locking, time-stamped versions, …

Abort-oriented vs. Locking control

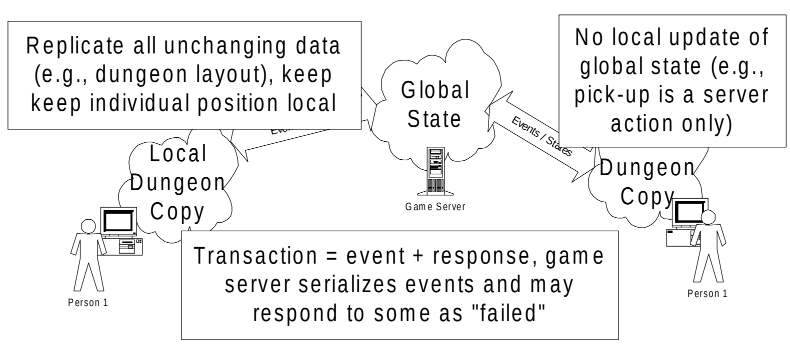
- Use abort-oriented control when
  - Locking (at the right level) is too expensive
    - e.g., in a multi-player game, the shared world should not be locked between messages
  - Aborting a transaction has no serious effects, OR
  - Abort/Retry can be hidden from user
- Use locking-oriented control when
  - Conflicts are rare, or blocking is acceptable
- Use more complex concurrency control when
  - there is no other acceptable choice

Where to Start?

- What consistency is needed?
  - Identify shared state that must be treated consistently
  - Include local state with implicit consistency, e.g., what must player A and B agree about?
- What operations must be serialized?
  - Balance simple consistency reasoning with acceptable performance
- Choose approach and granularity
  - Easiest if one approach throughout
    - mix and match is possible only under special conditions

Example: Networked Multi-Player Game

- Concurrency control handled at level of overall strategy, then applied (independently) to each module



Using Language Primitives

- Java: Monitors
  - "Synchronized" methods provide locking concurrency control at the level of individual objects
    - Adequate if method = transaction
    - Makes deadlock unlikely (but not impossible)
    - May not ensure global consistency; explicit locking may be necessary, but is *much* harder to design correctly
- Distributed processes
  - Remote procedure control
    - Event dispatch may provide monitor-like concurrency control
  - Explicit locking (e.g., Unix flock) also possible

Summary

- Concurrency
  - Main problems are races (lost update) and deadlock
  - Difficult to reason about all possible interleavings
- Concurrency control
  - Known (and verified) strategies for maintaining consistency
  - Much easier than reasoning directly about interleavings
  - From OS and database research, but widely applicable
- To apply: Identify consistency needs, then transactions, then strategy; then design details

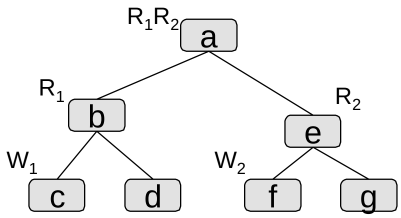
Supplementary Slides

- What are these?
  - These are details that don't fit in a one-hour lecture, but which you may find useful
    - Mostly as starting points. You'll need outside reading to get enough detail to actually use these techniques.
- What is here?
  - Other design rules for concurrent and real-time systems
  - Specialized and advanced concurrency control methods

Responsiveness and Priority

- Priority scheduling rules:
  - Assign highest priority to tasks with shortest periods or deadlines, rather than the most "urgent"
    - This is called "rate-monotonic" (or "deadline monotonic") scheduling, and it results in better response than ad hoc priority assignment based on urgency [Liu & Layland]
  - Avoid priority inversion
    - "Priority inversion" occurs when a high-priority task waits for a low-priority task
    - At a system level, low-priority tasks should inherit the priority of high-priority tasks waiting for the locks they hold
    - If system doesn't do it, simulate by using high-priority tasks to perform operations on objects locked by high-priority tasks
      - This is called "priority ceiling," and is essential to achieving worst-case timing requirements in hard real-time systems. See me if you want papers that describe the details.

Tree Locking



Process 1 locks a,b for reading, c for writing

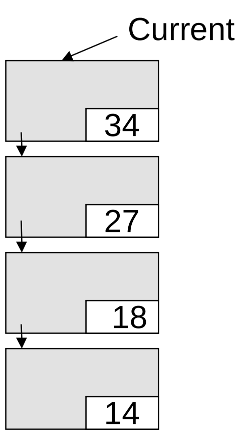
Process 2 locks a,e for reading, f for writing

The two updates can proceed concurrently, and the result will be as if they were serialized.

- If the global state is tree-structured, you can use that structure to improve locking performance
- Lock a "path" from root to the node to be locked
  - Always starting from the root
- Locks "above" changed node can be read locks
  - CREW protocol is the (only) source of performance improvement

Concurrency Control with Time-Stamps

See Bernstein et al to get the details (and to get it right, since I'm going from memory)



- Each transaction is initially given a time-stamp
  - They have to be properly ordered, but need not reflect "real" time
- Writing = creating a new version
  - Marked with the transaction time-stamp
- Transactions can "read from the past"
  - Transaction stamped 29 would read version marked 27, not current version
- Abort may be necessary
  - The past is read-only; cannot write a version older than the current