

Dealing with Deadlock (Review)

- *The Ostrich Approach* — stick your head in the sand and ignore the problem
- *Deadlock avoidance* — consider resources and requests, and only fulfill requests that will not lead to deadlock
 - ✗ Too hard for centralized systems, even harder in distributed systems!!
- *Deadlock prevention* — eliminate one of the 4 deadlock conditions
- *Deadlock detection and recovery* — detect, then break the deadlock
 - ✗ More difficult when state is distributed
 - Must avoid reporting false deadlock
- ➔ In distributed systems, we typically assume single resource instances

1

Spring 2000, Lecture 18

Deadlock Conditions (Review)

- These 4 conditions are **necessary** and **sufficient** for deadlock to occur:
 - **Mutual exclusion** — if one process holds a resource, other processes requesting that resource must wait until the process releases it (only one can use it at a time)
 - **Hold and wait** — processes are allowed to *hold* one (or more) resource and be *waiting* to acquire additional resources that are being held by other processes
 - **No preemption** — resources are released voluntarily; neither another process nor the OS can force a process to release a resource
 - **Circular wait** — there must exist a set of waiting processes such that P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2, ... Pn-1 is waiting for a resource held by Pn, and Pn is waiting for a resource held P0

2

Spring 2000, Lecture 18

Deadlock Prevention

- Basic idea: ensure that one of the 4 conditions for deadlock can not hold
- **Mutual exclusion** — if one process holds a resource, other processes requesting that resource must wait until the process releases it
 - Hard to avoid mutual exclusion for non-sharable resources
 - Printer & other I/O devices
 - Files
 - Network connections
 - However, many resources are sharable, so deadlock can be avoided for those resources
 - Read-only files
 - For printer, avoid mutual exclusion through spooling — then process won't have to wait on physical printer

3

Spring 2000, Lecture 18

Deadlock Prevention (cont.)

- **Circular wait** — there must exist a set of waiting processes such that P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2, ... Pn-1 is waiting for a resource held by Pn, and Pn is waiting for a resource held P0
 - To avoid, impose a total order on all resources, and require process to request resource in that order
 - Order: disk drive, printer, CDROM
 - Process A requests disk drive, then printer
 - Process B requests disk drive, then printer
 - Process B does not request printer, then disk drive, which could lead to deadlock
 - Order should be in the logical sequence that the resources are usually acquired
 - Allow process to release all resources, and start request sequence over
 - Or force process to request total number of each resource in a single request

4

Spring 2000, Lecture 18

Deadlock Prevention (cont.)

- **No preemption** — resources are released voluntarily; neither another process nor the OS can force a process to release a resource
 - To avoid, allow preemption
 - If process A requests resources that aren't available, see who holds those resources
 - If the holder (process B) is waiting on additional resources, preempt the resource requested by process A
 - Otherwise, process A has to wait
 - » While waiting, some of its current resources may be preempted
 - » Can only wake up when it acquires the new resources plus any preempted resources
 - If a process requests a resource that can not be allocated to it, **all** resources held by that process are preempted
 - Can only wake up when it can acquire all the requested resources
 - Only works for resources whose state can be saved/restored (memory, not printer)

5

Spring 2000, Lecture 18

Deadlock Prevention (cont.)

- **Hold and wait** — processes are allowed to *hold* one (or more) resource and be *waiting* to acquire additional resources that are being held by other processes
 - To avoid, ensure that whenever a process requests a resource, it doesn't hold any other resources
 - Request all resources (at once) at beginning of process execution
 - Process which loops forever?
 - Request all resources (at once) at any point in the program
 - To get a new resource, release all current resources, then try to acquire new one plus old ones all at once
 - Difficult to know what to request in advance
 - Wasteful; ties up resources and reduces resource utilization
 - Starvation is possible

6

Spring 2000, Lecture 18

Atomic Transactions

- A *transaction* (also called an *atomic transaction*) is a set of operations that perform some logically complete task (from the field of databases)
 - Transactions must be prevented from interfering with one another
 - If a transaction terminates normally, its effects are permanent; otherwise it has no effect
- Example transaction involving a client and three bank accounts A, B, and C:
 - Withdraw(A, 100);
 - Deposit(B, 100);
 - Withdraw(C, 200);
 - Deposit(B, 200);
- Result is \$100 transferred from A to B, and \$200 transferred from C to

7

Spring 2000, Lecture 18

ACID Properties of a Transaction (Härder and Reuter, 1983)

- **A**tomicity — a transaction is either performed in its entirety or not at all; it appears to an outside observer as a single, instantaneous, indivisible action
- **C**onsistency — a transaction must take the database from one consistent state to another; invariants that should always hold will hold after the transaction
- **I**solated (Serializable) — if two transactions run at the same time, the result must look as if they ran sequentially in some arbitrary order; a transaction's updates must not be visible to other transactions until it commits
- **D**urable — once a transaction commits, its result is permanent (must never be lost)

8

Spring 2000, Lecture 18

Other Properties and Implications of Atomic Transactions

- Recoverability — the changes due to all completed transactions must be available in permanent storage (write to permanent storage before reporting the transaction complete)
 - If a server halts unexpectedly, when it wakes up again it aborts any uncommitted transactions, and recovers data values committed by recent transactions
- Server is responsible for synchronizing operations to ensure that the isolation / serializability requirement is met
 - Simple but unacceptable — perform each transaction sequentially
 - Harder but generally required — interleave operations of various transactions, while ensuring that isolation holds

9

Spring 2000, Lecture 18

Transaction Primitives

- Begin transaction — start a transaction
- Operations
 - Read — read data from a file or object
 - Write — write data to a file or object
 - Others appropriate to the type of transaction...
- Commit and end transaction — save updates and terminate the transaction
 - Changes are permanently recorded; all future transactions will see the results of the changes made during the transaction
- Abort and end transaction — restore system state and terminate the transaction
 - None of the changes are visible in future transactions

10

Spring 2000, Lecture 18

Implementing Transactions, and Recovery from an Aborted Transaction

- Can't just update objects
 - Doesn't enforce atomicity
 - State can't be restored on abort
 - Multiple transactions will not be isolated
- When a process begins a transaction, give it a **private workspace**
 - Contains copies of all files and objects it needs
 - Changes are made to private copies
 - Commit changes originals, abort leaves originals untouched
 - Optimizations:
 - Don't copy objects read but not written
 - Copy only the file index (location of blocks on disk) and blocks actually written

11

Spring 2000, Lecture 18

Implementing Transactions, Recovery (cont.)

- Record changes in a **writeahead log**
 - Record in the writeahead log ("ahead" of the change)
 - Which transaction is making the change
 - Which file and block is being changed
 - Old and new values
 - *Immediate* update:
 - Operations record in log as described above, then update the actual data
 - If transaction aborts, must use log to *rollback* — restore original state
 - *Deferred* update:
 - Operations update local workspace
 - Commit writes record to log as described above, then updates the actual data
 - If transaction aborts, data remains unchanged
 - Log can also be used to recover from a crash (compare log to actual values to determine state at crash)

12

Spring 2000, Lecture 18

Need for Concurrency Control

- *Concurrency control* — allow two or more transactions to proceed concurrently, while preserving serializability (isolation)

- Lost update problem:

- Account A = \$100, B = \$200, C = \$300
 - Transaction T transfers \$4 from A to B
 - Transaction U transfers \$3 from C to B
 - Should end A = \$96, B = \$207, C = \$297

- U's update of B is lost:

<u>Transaction T</u>		<u>Transaction U</u>	
bal=read(A)	\$100		
write(A, bal-4)	\$96		
		bal=read(C)	\$300
		write(C, bal-3)	\$297
bal=read(B)	\$200		
		bal=read(B)	\$200
		write(B, bal+3)	\$203
write(B, bal+4)	\$204		

Need for Concurrency Control (cont.)

- Inconsistent retrievals problem:

- Account A = \$200, B = \$200
 - Transaction T transfers \$100 from A to B
 - Transaction U computes sum of all accounts in the bank
 - Should end A = \$100, B = \$300, total = \$400+
- U's retrievals are inconsistent because T has not completed the transfer when the sum is calculated:

<u>Transaction T</u>		<u>Transaction U (part)</u>	
bal=read(A)	\$200		
write(A, bal-100)	\$100		
		bal=read(A)	\$100
		bal+=read(B)	\$300
bal=read(B)	\$200		
write(B, bal+100)	\$300		

Concurrency Control — Enforcing Serializability

- Lost update problem:

- Not interleaving updates:

<u>Transaction T</u>		<u>Transaction U</u>	
bal=read(A)	\$100		
write(A, bal-4)	\$96		
		bal=read(C)	\$300
		write(C, bal-3)	\$297
bal=read(B)	\$200		
write(B, bal+4)	\$204		
		bal=read(B)	\$204
		write(B, bal+3)	\$207

- Inconsistent retrievals problem:

- Not interleaving transfer retrieval:

<u>Transaction T</u>		<u>Transaction U (part)</u>	
bal=read(A)	\$200		
write(A, bal-100)	\$100		
bal=read(B)	\$200		
write(B, bal+100)	\$300		
		bal=read(A)	\$100
		bal+=read(B)	\$400