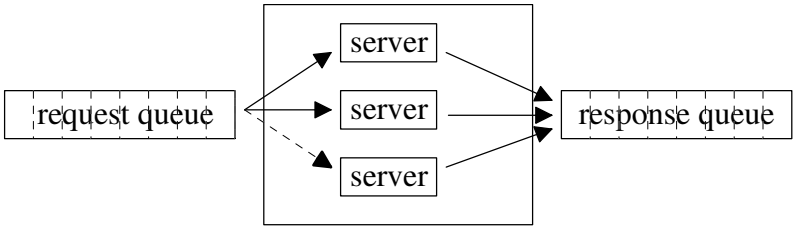
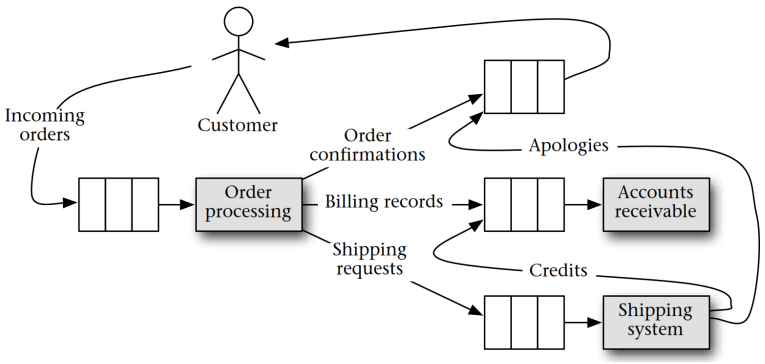
1. *Message-Queuing Systems* - middleware communication services that support atomic transaction concept



Each server:
- dequeue request
- process request
- enqueue response

Atomic Transaction Behavior:
- If any processing step fails, request left in request queue for a retry
- No request lost or visible signs of retries (e.g., no duplicate responses, etc.)
- provides durability (e.g., system crash & restart does not effect each request generating one response)

More advanced workflow systems might have several processing stages connected by intermediate message queues. Each processing stage is treated as a separate transaction, so the whole workflow might not exhibit atomicity.

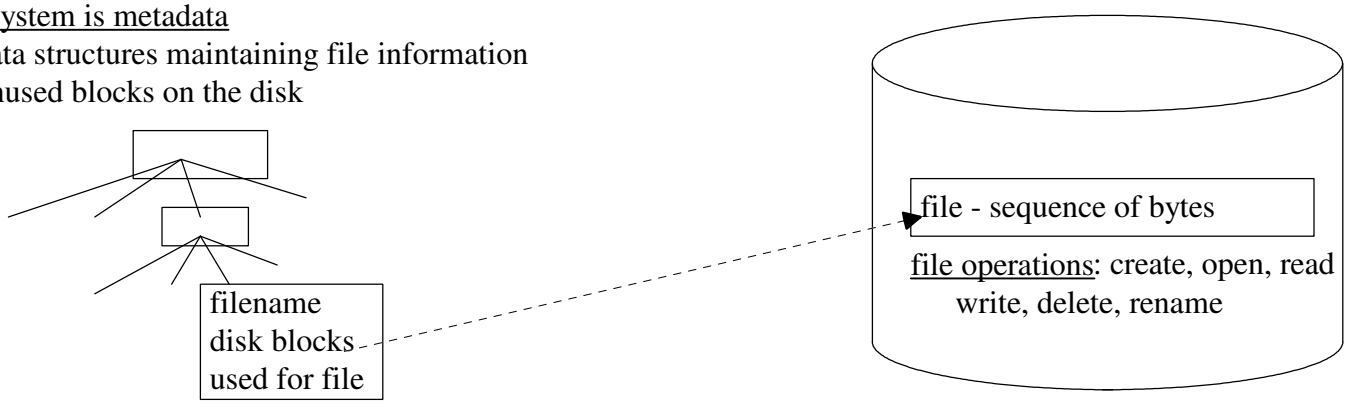


a) In the example of an ordered item being out-of-stock, how might the customer observe that the whole process in not atomic?

b) How does treating each workflow stage as an atomic transaction (instead of one whole atomic transaction) reduce the design's "cognitive burden"?

2. Journaled File System - OS's file system employs atomic transaction to at least maintain structural consistency of the file system (e.g., NTFS (Windows), HFS Plus (Mac OS X), ext3fs, reiserfs, JFS, XFS (Linux/UNIX))

File system is metadata
- data structures maintaining file information
- unused blocks on the disk



a) A file-system operation can involve several updates to information in persistent storage. Consider appending to a file. What file-system information must be modified?

b) Why do file system operations need to be atomic and durable transactions?

c) Why might the OS "bundle" the transaction commits for a minute or two in memory with only periodic updates to persistent storage?

d) What penalty would a system crash cause if the OS "bundles" the transaction commits?

f) Often file systems protect only its metadata with application data files left in an inconsistent state after a crash. However, some file systems (e.g., Transactional NTFS (TxF) for Windows Vista) support transactions by allowing application programs to group updates into a transaction. Why might is be especially useful for programs with transactions that span many files and directories?

3. Mechanisms to Ensure Atomicity:
- two-phase locking - isolate concurrent transactions from each other
- undo log - assures that failed transactions have no visible effect

Concurrent transaction execution must be equivalent to some serial execution of these transactions - *serializable*. *Equivalence/serializability* definition of text simplest to justify two-phase locking. (More advanced concurrency control mechanisms need more general definition)

Each transaction executes a sequence of actions that read or write some stored entity and those actions lock or unlock a readers/writers lock. Each entitiy has its own lock. Notations:
$r_j(x)$ or $r_j(x, v)$ - read of entity x by transaction T_j getting value v
$w_j(x)$or $w_j(x, v)$ - write of value of v to entity x by transaction T_j
$s_j(x)$ - acquisition of shared (reader) lock on entity x by transaction T_j
$e_j(x)$ - acquisition of exclusive (writer) lock on entity x by transaction T_j
$\overline{s_j}(x)$ - unlock of shared (reader) lock on entity x by transaction T_j
$\overline{e_j}(x)$ - unlock of exclusive (writer) lock on entity x by transaction T_j
$u_j(x)$ - upgrade by transaction T_j of its held shared (reader) lock on entity x to an exclusive (writer) lock
A *history* is the sequence of actions of execution. A *system's history* is a time-ordered interleaving of all the transactions' histories. Locking actions are shown when a lock is granted (not requested). If a transaction aborts and does some extra writes to undo the effects of earlier writes, then those undo writes must be explicitly listed in the history. We'll assume that transactions have no effects other than storage (i.e., no I/O).

Suppose a system with two variables *x* and *y* having an invariant of x = y, and both initial start with the value 5. Consider two transactions: T_1 which adds 3 to both *x* and *y*, and T_2 which doubles both *x* and *y*. Both transactions preserves the invariant that x = y. Now, consider the system history of serially execution of T_1 followed by T_2:

$e_1(x), r_1(x,5), w_1(x,8), \overline{e_1}(x), e_1(y), r_1(y,5), w_1(y,8), \overline{e_1}(y), e_2(x), r_2(x,8), w_2(x,16), \overline{e_2}(x), e_2(y), r_2(y,8), w_2(y,16), \overline{e_2}(y)$

a) Write the other serial history in which T_2 is followed by T_1.

b) How do the resulting values of x and y compare after each of the above serial executions?

c) Why is the following history of concurrently execution of T_1 and T_2 serializable?

$e_1(x), r_1(x,5), w_1(x,8), \overline{e_1}(x), e_2(x), r_2(x,8), w_2(x,16), \overline{e_2}(x), e_1(y), r_1(y,5), w_1(y,8), \overline{e_1}(y), e_2(y), r_2(y,8), w_2(y,16), \overline{e_2}(y)$

d) Write a history of concurrently execution of T_1 and T_2 that is nonserializable. You cannot change the steps within either transaction only their interleaving.

e) Goal of two-phase locking is to use locks in a disciplined fashion to rule out nonserializable histories. We'll also assume that our transactions don't contain trivial errors. Complete each of the following:

i. Pair every lock operation with the corresponding _____.
ii. If the transaction is holding the lock, it will not try to _____.
iii. The exception to ii. being _____.
iv. A transaction will not try to unlock _____.
v. At the end of a transaction, all locks will be left _____.

We'll also assume that locks function correctly. Complete each of the following:

i. No transaction will ever be granted a lock in shared mode while _____.
ii. No transaction will ever be granted a lock in exclusive mode while _____.

f) Formally, two histories are equivalent if the first history can be turned into the second by performing a succession of equivalence-preserving swaps which reverses the order of adjacent actions, subject to the following constraints:

- The two actions must be from different transactions. Why?
- The two actions must not be any of the following seven *conflicting* pairs (i.e., *forbidden swaps*):

	Forbidden Swap	Explain why the swap is forbidden
1	, $\overline{e_j}(x), s_k(x)$	
2	, $\overline{e_j}(x), e_k(x)$	
3	, $\overline{s_j}(x), e_k(x)$	
4	, $\overline{s_j}(x), u_k(x)$	
5	$w_j(x), r_k(x)$	
6	$r_j(x), w_k(x)$	
7	$w_j(x), w_k(x)$	

g) Why would any swap involving two different entities by different transactions be equivalence-preserving?

h) Why would swapping reads of the on the same entity by two different transactions be equivalence-preserving?

4. A transaction obeying the following *two-phase locking rules* suffices to ensure serializability:

- For any entity that it operates on, the transaction locks the corresponding lock exactly once, sometime before it reads or writes the entity the first time, and unlocks it exactly once, sometime after it reads or writes the entity the last time.
- For any entity the transaction writes into, either the transaction initially obtains the corresponding lock in exclusive mode, or it upgrades the lock to exclusive mode sometime before writing.
- The transaction performs all its lock and upgrade actions before performing any of its unlock actions.

For each of the following transactions on x and y, determine whether two-phase locking rules are obeyed.

a) $e_1(x), r_1(x), w_1(x), e_1(y), \overline{e_1}(x), r_1(y),, w_1(y),\overline{e_1}(y)$

b) $e_1(x), r_1(x), w_1(x), \overline{e_1}(x), e_1(y), r_1(y), w_1(y),\overline{e_1}(y)$

c) $s_1(x), r_1(x), u_1(x), w_1(x),s_1(y), r_1(y), u_1(y), w_1(y), \overline{e_1}(x), \overline{e_1}(y)$

5. To make programming transactions simpler, a system could automatically insert locks and unlock actions which follow the two-phase rules by

- Immediately before any read action, acquire
- Immediately before any write action, acquire
- At the very end of the transaction, unlock