PRINCIPLES OF COMPUTER SYSTEM DESIGN

CHAPTER 9

ATOMICITY: ALL-OR-NOTHING AND ISOLATION

JANUARY 2007

TABLE OF CONTENTS

Glossary

abort—Upon deciding that an all-or-nothing action cannot or should not commit, to undo all of the changes previously made by that all-or-nothing action. After aborting, the state of the system, as viewed by anyone above the layer that implements the all-or-nothing action, is as if the all-or-nothing action never existed. Compare with *commit*.

action—An operation performed by an interpreter. Examples include a microcode step, a machine instruction, a higher-level language instruction, a procedure invocation, a shell command line, a response to a gesture at a graphical interface, or a database update.

all-or-nothing (adj.); all-or-nothing atomicity (n.)—A property of a multi-step action that if an anticipated failure occurs during the steps of the action, the action will appear above the layer that implements it either never to have started or else to have been accomplished completely. Compare with *isolated*.

archive—A record, usually kept in the form of a log, of old data values, for auditing, recovery from application mistakes, or historical interest.

atomic (adj.); atomicity (n.)—A property of a multi-step action that there be no evidence that it is composite above the layer that implements it. An atomic action can be *isolated*, which means that its internal composition is not evident to other concurrent threads. An atomic action can also be *all-or-nothing*, which means that if an anticipated failure occurs during the action, the action will appear to higher layers either never to have started or else to have completed successfully. An atomic action that is *both* all-or-nothing and isolated is known as a *transaction*.

atomic storage—Cell storage for which a multi-cell PUT can have only two possible outcomes: 1) it stores all data successfully, or 2) it does not change the previous data at all. In consequence, either a concurrent thread or (following a failure) a later thread doing a GET will always read either all old data or all new data. Computer architectures that provide non-atomic multi-cell PUTS are said to be subject to *write tearing*.

blind write—An update to a data value *X* by a transaction that did not previously read *X*.

cell storage—Storage in which a WRITE or PUT operates by overwriting, thus destroying previously stored information. Many physical storage devices, including magnetic disk and CMOS random access memory, implement cell storage. Compare with *journal storage*.

checkpoint—1. (n.) Information written to non-volatile storage that is intended to speed up recovery from a crash. 2 (v.) To write a checkpoint.

commit—To renounce the ability to abandon an all-or-nothing action unilaterally. One usually commits an all-or-nothing action before making its results available to concurrent or later all-or-nothing actions. Before committing, the all-or-nothing action can be abandoned and one can pretend that it had never been undertaken. After committing, the all-or-nothing action must be able to complete. A committed all-or-nothing action cannot be abandoned; if it can be determined precisely how far its results have propagated, it may be possible to reverse some or all of its effects by *compensation*. Commitment also usually includes an expectation that the results preserve any appropriate invariants and will be durable to the extent that the application requires those properties. Compare with *abort*.

compensate (adj.); compensation (n.)—To perform an action that reverses the effect of some previously committed action. Compensation is intrinsically application dependent; it is easier to reverse an incorrect accounting entry than it is to undrill an unwanted hole.

do action—(n.) Term used in some systems for a *redo action*.

exponential random backoff—When interference between two actions requires a retry, the technique of introducing a random delay before retrying, in the hope that with different relative timing the interference will not recur. The qualifier “exponential” means that if an action repeatedly encounters interference, it repeatedly multiplies (typically by 2) the size of the interval from which it randomly chooses its next delay.

force—(v.) When output may be buffered, to ensure that a previous output value has actually been written to durable storage or sent as a message. Caches that are not write-through usually have a feature that allows the invoker to force some or all of their contents to the secondary storage medium.

install—In a system that uses logs to achieve all-or-nothing atomicity, to write data to *cell storage*.

isolated (adj.); isolation atomicity (n.)—A property of a multi-step action that above the layer that implements it, concurrent threads cannot discover that it is composite. Compare with *all-or-nothing atomicity* and *atomic*.

journal storage—Storage in which a WRITE or PUT appends a new value, rather than overwriting a previously stored value. Compare with *cell storage*.

lock—A flag associated with a data object, set by an action to warn other, concurrent actions, that the object is currently in use and that it may be a mistake for other actions to read or write it. Locks are one technique used to achieve *isolation*.

lock point—In a system that provides isolation by locking, the first instant in an isolated action when every lock that will ever be in its lock set has been acquired.

lock set—The collection of all locks acquired during the execution of an isolated action.

log—1. (n.) A specialized use of *journal storage* to maintain an append-only record of some application activity. Logs are used to implement all-or-nothing actions, for performance enhancement, for archiving, and for reconciliation. 2. (v.) To append an record to a log.

logical locking—Locking of higher-layer data objects such as records or fields of a data base. Compare with *physical locking*.

mark point—1. (adj.) An isolation discipline in which each newly created isolated action *n* must wait to begin reading shared data objects until action *(n – 1)* has marked all of the variables it intends to modify. 2. (n.) The instant at which an isolated action has marked all of the variables it intends to modify.

optimistic concurrency control—A concurrency control scheme that allows concurrent threads to proceed even though there may be a risk that they will interfere with each other, with the plan of detecting if there actually is interference and if necessary forcing one of the threads to abort and retry. Optimistic concurrency control is an effective technique in situations where interference is possible but not likely. Compare with *pessimistic concurrency control*.

pending—A state of an all-or-nothing action, when that action has not yet either committed or aborted. Also used to describe the value of a variable that was set or changed by a still-pending all-or-nothing action.

pessimistic concurrency control—A concurrency control scheme that forces a thread to wait if there is any chance that by proceeding it may interfere with another, concurrent, thread. Pessimistic concurrency control is an effective technique in situations where interference between concurrent threads has a high probability. Compare with *optimistic concurrency control*.

physical locking—Locking of lower-layer data objects, typically chunks of data whose extent is determined by the physical layout of a storage medium. Examples of such chunks are disk sectors or even an entire disk. Compare with *logical locking*.

prepared—In a layered or multiple-site all-or-nothing action, a state of a component action that has announced that it can, on command, either commit or abort. Having reached this state, it awaits a decision from the higher-layer coordinator of the action.

progress—A desirable guarantee provided by an isolation mechanism: that despite potential interference from concurrency some useful work will be done. An example of such a guarantee is that the isolation mechanism will not abort at least one member of the set of concurrent actions. In practice, lack of a progress guarantee can sometimes be repaired by using *exponential random backoff*. In formal analysis of systems, progress is one component of a property known as “liveness”. Progress is an assurance that the system will move toward some specified goal, while liveness is an assurance that the system will eventually reach that goal.

redo action—(n.) An application-specified action that, when executed during failure recovery, produces the effect of some committed component action whose effect may have been lost in the failure. (Some systems call this a “do action”. Compare with *undo action.)*

rollback recovery—(Also known as *undo logging*.) A *write-ahead log* protocol with the additional requirement that the application perform all *install* actions *before* logging an outcome record. If there is a failure before the all-or-nothing action commits, a recovery procedure can use the log record to undo the partially completed all-or-nothing action. Compare with *roll-forward recovery*.

roll-forward recovery—(Also known as *redo logging*.) A *write-ahead log* protocol with the additional requirement that the application log its outcome record *before* it performs any install actions. If there is a failure before the all-or-nothing action passes its commit point, the recovery procedure does not need to undo anything; if there is a failure after commit, the recovery procedure can use the log record to ensure that cell storage installs are not lost. Compare with *rollback recovery*.

serializable—A property of isolated actions, that even if several operate concurrently, the result is the same as if they had acted one at a time, in some sequential (in other words, serial) order.

shadow copy—A working copy of an object that an all-or-nothing action creates so that it can make several changes to the object and still maintain all-or-nothing atomicity, because the original remains unmodified. When the all-or-nothing action has made all of the changes, it then carefully exchanges the working copy with the original, thus preserving the appearance that all of the changes occurred atomically. Depending on the implementation, either the original or the working copy may be identified as the “shadow” copy, but the technique is the same in either case.

simple locking—A locking protocol for isolation of atomic actions that requires that no data be read or written except during the *lock point*. For the atomic action to also be *all-or-nothing*, a further requirement is that commit (or abort) occur during the lock point. Compare with *two-phase locking*.

simple serialization—An isolation protocol that requires that each newly created atomic action must wait to begin execution until all previously defined atomic actions are no longer pending.

transaction—a multistep action that is both atomic in the face of failure and atomic in the face of concurrency. That is, it is both *all-or-nothing* and *isolated*.

two generals dilemma—An intrinsic problem that no finite protocol can guarantee to simultaneously coordinate state values at two places that are linked by an unreliable communication network.

two-phase commit—A protocol that creates a higher-layer transaction out of separate, lower-layer transactions. The protocol first goes through a *preparation* (sometimes called *voting*) *phase*, at the end of which each lower-layer transaction reports either that it cannot perform its part or that it is prepared to either commit or abort. It then enters a *commitment phase*, in which the higher-layer transaction, acting as a coordinator, makes a final decision; thus the name two-phase. Two-phase commit has no connection with the similar-sounding term *two-phase locking*.

two-phase locking—A locking protocol for isolation of atomic actions that requires that no locks be released until all locks have been acquired (that is, there must be a *lock point*). For the atomic action to also be *all-or-nothing*, a further requirement is that no locks for objects to be written be released until the action commits. Compare with *simple* *locking*. Two-phase locking has no connection with the similar-sounding term *two-phase commit*.

undo action—An application-specified action that, when executed during failure recovery or an abort procedure, reverses the effect of some previously performed, but not yet committed, component action. The goal is that neither the original action nor its reversal be visible above the layer that implements the action. (Compare with *redo* and *compensate*.)

version history—The set of all values for an object or variable that have ever existed, stored in *journal storage*.

write-ahead-log (WAL) protocol—A recovery protocol that requires appending a log record in *journal storage* before installing the corresponding data in *cell storage*.

Overview

This chapter explores two closely related system engineering design strategies. The first is *all-or-nothing atomicity*, a design strategy for masking failures that occur while interpreting programs. The second is *isolation atomicity*, a design strategy for coordinating concurrent activities. Chapter 8 introduced failure masking, but did not show how to mask failures of running programs. Chapter 5 introduced coordination of concurrent activities, and presented solutions to several specific problems, but it did not explain any systematic way to assure isolation. This chapter explores ways to systematically synthesize a design that provides both failure masking and isolation.

Many useful applications can benefit from atomicity. For example, suppose that you are trying to buy a toaster from an Internet store. You click on the button that says "purchase", but before you receive a response the power fails. You would like to have some assurance that, despite the power failure, either the purchase went through properly or that nothing happen at all. You don't want to find out later that your bank account was charged but the Internet store didn't receive word that it was supposed to ship the toaster. In other words, you would like to see that the action initiated by the "purchase" button be all-or-nothing despite the possibility of failure. And if the store has only one toaster in stock and two customers both click on the "purchase" button for a toaster at about the same time, one of the customers should receive a confirmation of the purchase, and the other should receive a "sorry, out of stock" notice. It would be problematic if both customers received confirmations of purchase. In other words, both customers would like to see that the action initiated by the "purchase" button be correctly isolated from other, concurrent purchase actions.

The single conceptual framework of atomicity provides a powerful way of thinking about both all-or-nothing failure masking and isolation of concurrent activities. *Atomicity* is the performing of a sequence of steps, called *actions*, so that they appear to be done as a single, indivisible step, known in operating system and architecture literature as an *atomic action* and in database management literature as a *transaction*. When a fault causes a failure in the middle of a correctly designed atomic action, it will appear to the invoker of the atomic action that the atomic action either completed successfully or did nothing at all. Similarly, when several atomic actions are going on concurrently, each atomic action will appear to take place either completely before or completely after each other atomic action. Together, all-or-nothing atomicity and isolation atomicity provide a particularly strong form of modularity: they hide the fact that the atomic action is actually be composed of multiple steps.

The result is a <u>*sweeping simplification*</u> in the description of the possible states of a system. This simplification provides the basis for a methodical approach to recovery from failures and coordination of concurrent activities that simplifies design, simplifies understanding for later maintainers, and simplifies verification of correctness. These desiderata are particularly important, because errors caused by mistakes in coordination usually depend on the relative timing of external events and among different threads. When a timing-dependent error occurs, the difficulty of discovering and diagnosing it can be orders of magnitude greater than that of finding a mistake in a purely sequential activity. The reason is that even a small number of concurrent activities can have a very large number of potential real time sequences. It is usually impossible to determine which of those many potential sequences of steps preceded the error, so it is effectively impossible to reproduce the error under more carefully controlled circumstances. Since debugging this class of error is so hard, techniques that assure correct coordination are particularly valuable.

The remarkable thing is that the same systematic approach—atomicity—to failure recovery also applies to coordination of concurrent activities. In fact, since one must be able to deal with failures while at the same time coordinating concurrent activities, any attempt to use different strategies for these two problems requires that the strategies be compatible. Being able to use the same strategy for both is another *sweeping simplification*.

Atomic actions are a fundamental building block that is widely applicable in computer system design. Atomic actions are found in database management systems, in register management for pipelined processors, in file systems, in change-control systems used for program development, and in many everyday applications such as word processors and calendar managers.

The sections of this chapter define atomicity, examine some examples of atomic actions, and explore systematic ways of achieving atomicity: *version histories*, *logging*, and *locking protocols*. Chapter 10 then explores some applications of atomicity. Case studies in the appendices of both chapters provide real-world examples of atomicity as a tool for creating useful systems.

Sidebar 9–1: Actions and transactions

The terminology used by system designers to discuss atomicity can be confusing, because the concept was identified and developed independently by database designers and by hardware architects.

An action that changes several data values can have any or all of at least four independent properties: it can be *all-or-nothing* (either all or none of the changes happen), it can be *isolated* (the changes all happen either before or after every concurrent action), it can be *constraint-maintaining* (the changes maintain some specified invariant), and it can be *durable* (the changes last as long as they are needed).

Designers of database management systems customarily are concerned only with actions that are both all-or-nothing and isolated, and they describe such actions as *transactions*. In addition, they use the term *atomic* primarily in reference to all-or-nothing atomicity. On the other hand, hardware processor architects customarily use the term *atomic* to describe an action that exhibits isolation atomicity.

This book does not attempt to change these common usages. Instead, it uses the qualified terms “all-or-nothing atomicity” and “isolation atomicity.” The unqualified term “atomic” may imply all-or-nothing, or isolated, or both, depending on the context. The text uses the term “transaction” to mean an action that is *both* all-or-nothing and isolated.

All-or-nothing atomicity and isolation atomicity are universally defined properties of actions, while constraints are properties that different applications define in different ways. Durability lies somewhere in between, because different applications have different durability requirements. At the same time, implementations of constraints and durability usually have a prerequisite of atomicity. Since the atomicity properties are modularly separable from the other two, this chapter focuses just on atomicity. Chapter 10 then explores how a designer can use transactions to implement constraints and enhance durability.

A. Atomicity

1. Interrupts revisited

Recall from chapters 2 and 5 that a thread normally performs actions one after another, as directed by the instructions of the current program, but that certain events may catch the attention of the thread's interpreter, causing the interpreter, rather than the program, to supply the next instruction. When such an event happens, a different program, running in an interrupt thread, takes control. The event may be a signal arriving from outside the interpreter or it may be a problem that the interpreter has detected in the interrupted thread.

The source of the interrupt is less interesting than what the interrupt thread does next. The simplest case, as described on page 5–86, is that the interrupt thread just calls ADVANCE to alert some other thread that an event it has been waiting for has happened, for example because an I/O operation has now completed. In this case, we are interested in isolating the action of the interrupt thread and of any threads it alerts from the action of the original thread.

Alternatively, if the reason for the interrupt is that something has happened within the thread that was currently running, the interrupt event invokes an exception handler, which runs in the environment of the original thread. Sidebar 9–2 offers some examples. The exception handler either adjusts the environment to eliminate some problem (such as a missing page) so that the original thread can continue, or it declares that the original thread has failed and terminates it. In either case, the handler is likely to be interested in the state of the action that the original thread was performing at the instant of the interruption—was that action finished, or is it in a partially done state?

Ideally, the handler would like to see an all-or-nothing report: either the instruction that caused the exception completed or it didn't do anything. An all-or-nothing report means that the state of the original thread is described entirely with values belonging to the layer in which the handler is running. An example of such a value is the program counter, which identifies the next instruction that the thread is to execute. An in-the-middle report would mean that the state description involves values of a lower layer, probably the operating system or the hardware processor itself. In that case, knowing the next instruction is only part of the story; the handler would also need to know which parts of the current instruction were executed and which were not. An example might be an instruction that increments an address register, retrieves the data at the address now in that address register, and adds that value to the value in another register. If retrieving the data causes a missing-page exception, the description of the current state is that the address register has been incremented but the retrieval and addition have not yet been performed. Such an in-the-middle report can be problematic, because when the missing page has been retrieved the handler cannot simply

Sidebar 9–2: Events that might lead to invoking an exception handler:

1. A hardware fault occurs:

- The processor detects a memory parity fault.
- A sensor reports that the electric power has failed; the energy left in the power supply may be just enough to perform a graceful shutdown.

2. A hardware or software interpreter encounters something in the program that is clearly wrong:

- The program tried to divide by zero.
- The program supplied a negative argument to a square root function.

3. Continuing requires some resource allocation or deferred initialization:

- The running thread encountered a missing-page exception in a virtual memory system.
- The running thread encountered an indirection exception, indicating that it encountered an unresolved procedure linkage in the current program.

4. More urgent work needs to take priority, so the user wishes to terminate the thread:

- This program is running much longer than expected.
- The program is running normally, but the user suddenly realizes that it is time to catch the last train home.

5. The user realizes that something is wrong and decides to terminate the thread:

- Calculating *e*, the program starts to display 3.1415…
- The user asked the program to copy the wrong set of files.

6. Deadlock:

- Thread A has acquired the scanner, and is waiting for memory to become free; thread B has acquired all available memory, and is waiting for the scanner to be released. Either the system notices that this set of waits cannot be resolved or, more likely, a timer that should never expire eventually expires. The system or the timer signals an exception to one or both of the deadlocked threads.

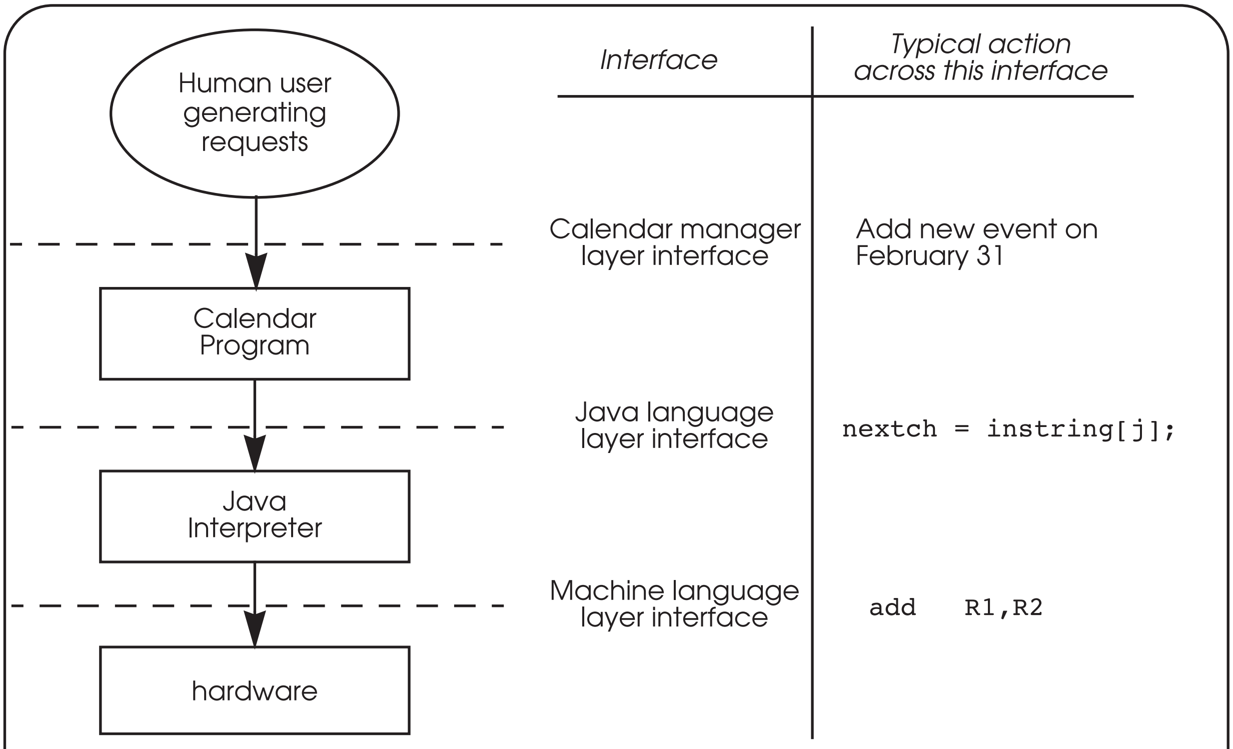
tell the processor to jump to the instruction that failed—that would increment the address register again, which is not what the programmer expected. Jumping to the next instruction isn't right, either, because that would omit the step of addition. An all-or-nothing report is preferable because it avoids the need for the handler to peer into the details of the next lower layer.

Sections A.2 and A.3 explore the case in which the exception terminates the running thread, thus creating a fault. Section A.4 examines the case in which the interrupted thread continues, oblivious (one hopes) to the interruption.

2. *Containing faults with layers*

The challenge presented by a fault in a running program is that at the instant of the fault, the program is typically in the middle of doing something, and it is usually not acceptable to leave things half-done. Our goal is to obtain a more graceful response, and the method will be to require that some sequence of actions behave as an atomic action. Atomic actions are closely related to the modularity that arises when things are organized in layers. Layered components have the feature that a higher layer can completely hide the existence of a lower layer. This hiding feature makes layers exceptionally effective at error containment and for systematically responding to faults.

To see why, recall the layered structure of the calendar management program of chapter 2, reproduced in figure 9–1 (that figure may seem familiar—it is a copy of figure 2–7 on page 2–23). The calendar program implements each request of the user by executing a sequence of Java language statements. Ideally, the user will never notice any evidence of the



Interface	*Typical action across this interface*
Calendar manager layer interface	Add new event on February 31
Java language layer interface	`nextch = instring[j];`
Machine language layer interface	`add R1,R2`

Figure 9–1: An application system with three layers of interpretation. The user has requested an action that will fail, but the failure will be discovered at the lowest layer. A graceful response involves atomicity at each interface.

composite nature of the actions implemented by the calendar manager. Similarly, each statement of the Java language is implemented by several actions at the hardware layer. Again, if the Java interpreter is carefully implemented, the composite nature of the implementation in terms of machine language will be completely hidden from the Java programmer.

Now consider what happens if the hardware processor detects a condition that should be handled as an exception—for example, a register overflow. The machine is in the middle of interpreting an action at the machine language layer interface—an ADD instruction somewhere in the middle of the Java interpreter program. That ADD instruction is itself in the middle of interpreting an action at the Java language interface—a Java expression to scan an array. That Java expression in turn is in the middle of interpreting an action at the user interface—a request from the user to add a new event to the calendar. The report "Overflow exception caused by the ADD instruction at location 41574" is not intelligible to the user at the user interface; that description is meaningful only at the machine language interface. Unfortunately, the implication of being "in the middle" of higher-layer actions is that the only accurate description of the current state of affairs is in terms of the progress of the machine language program.

The actual state of affairs in our example as understood by an all-seeing observer might be the following: the register overflow was caused by adding one to a register that contained a two's complement negative one at the machine language layer. That machine language add instruction was part of an action to scan an array of characters at the Java layer and a zero means that the scan has reached the end of the array. The array scan was embarked upon by the Java layer in response to the user's request to add an event on February 31. The highest-level interpretation of the overflow exception is "You tried to add an event on a non-existent date". We want to make sure that this report goes to the end user, rather than the one about register overflow. In addition, we want to be able to assure the user that this mistake has not caused an empty event to be added somewhere else in the calendar or otherwise led to any other changes to the calendar. Since the system couldn't do the requested change it should do nothing but report the error. Either a low-level error report or muddled data would reveal to the user that the action was composite.

In technical terms, if we look at the interface between two layers, we want the fault detected by the lower layer to be contained in a particular way: the sequence of actions of the lower layer should have *all-or-nothing atomicity.* All-or-nothing atomicity is defined as follows:

All-or-nothing atomicity

A sequence of steps is an *all-or-nothing action* if, from the point of view of its invoker, the sequence always either

- ***completes*,**

or

- **aborts in such a way that it appears that the sequence had never been undertaken in the first place. That is, it *backs out.***

Brief version: *Do it all or not at all.*

This form of atomicity is framed in terms of an atomic action dealing with the higher-layer client that invoked it. We shall soon discover that a different form of atomicity seems to be needed to deal with concurrent actions, and then we shall establish that the two forms are closely related.

In a multilayer interpreter, the idea is to design each of the actions of each layer to be all-or-nothing. That is, whenever an action of a layer is carried out by a sequence of actions of the next lower layer, the action either completes what it was asked to do or else it backs out, acting as though it had not been invoked at all. When control returns to a higher layer after a lower layer detects a fault, the problem of being "in the middle" of an action thus disappears.

In our calendar management example, we might expect that the machine language layer would complete the add instruction but signal an overflow exception; the Java interpreter layer would, upon receiving the overflow exception might then decide that its array scan has ended, and return a report of "scan complete, value not found" to the calendar management layer; the calendar manager would take this not-found report as an indication that it should back up, completely undo any tentative changes, and tell the user that the request to add an event on that date could not be accomplished because the date does not exist.

Thus some layers run to completion, while others back out and act as though they had never been invoked, but either way the actions are all-or-nothing. In this example, the failure would probably propagate all the way back to the human user to decide what to do next. A different failure might be intercepted by some intermediate layer that knows of a way to mask it. In that case, the all-or-nothing requirement is that the layer that masks the failure find that the layer below has either never started what was to be the current action or else it has completed the current action but has not yet undertaken the next one.

All-or-nothing atomicity is not usually achieved casually, but rather by careful design and specification. Designers often get it wrong. An unintelligible error message is the typical symptom that a designer got it wrong. To gain some insight into what is involved, let us examine some examples.

3. Examples of actions with and without the all-or-nothing property

One example of an all-or-nothing action is sometimes provided in the bus interface between a processor and a memory module. The bus may provide a "hold" line that, when set high by a processor, prevents all other active bus participants from using the bus. The purpose of this feature is to allow a processor to implement the read-and-set-memory instruction described on page 5–28, which involves reading a value from memory and writing another value back, without any other processor or input/output device reading or changing that same memory location in the interim. The processor designer usually takes some special precaution to avoid faults while performing one of these all-or-nothing read-rewrite operations. For example, if interrupts may be signalled by input/output devices, the processor may defer recognizing them until the end of the all-or-nothing action. Thus the operation of reading and rewriting the memory value is guaranteed, once undertaken, to run to the end. Alternatively, if the all-or-nothing action attempts to read from a nonexistent or unmapped address, it is able to abort without having modified any registers or locations in memory, and thus appear as though the action were never performed at all.

A second example, this time of an interface problem, is that actions that lack the all-or-nothing property are often discovered upon adding multilevel memory management to a computer architecture, especially to a processor that is highly pipelined. In this case, the interface that needs to be all-or-nothing lies between the processor and the operating system. Unless the original machine architect designed the instruction set with missing-page exceptions in mind, there may be cases in which a missing-page exception can occur "in the middle" of an instruction, after the processor has overwritten some register or after later instructions have entered the pipeline. When such a situation arises, the later designer who is trying to add the multilevel memory feature is trapped. The instruction cannot run to the end because one of the operands it needs is not in real memory. While the missing page is being retrieved from secondary storage, the designer would like to allow the operating system to use the processor for something else (perhaps even to run the program that fetches the missing page), but reusing the processor requires saving the state of the currently executing program, so that it can be restarted later when the missing page is available. The problem is how to save the next-instruction pointer.

If every instruction is an all-or-nothing action, the operating system can simply save as the value of the next-instruction pointer the address of the instruction that encountered the missing page. The resulting saved state description shows that the program is between two instructions, one of which has been completely executed, and the next one of which has not yet begun. Later, when the page is available, the operating system can restart the program by reloading all of the registers and setting the program counter to the place indicated by the next-instruction pointer. The processor will continue, starting with the instruction that previously encountered the missing page exception; this time it should succeed. On the other hand, if even one instruction of the instruction set lacks the all-or-nothing property, when an interrupt happens to occur during the execution of that instruction it is not at all obvious how the operating system can save the processor state for a future restart. Designers have come up with several techniques to retrofit the all-or-nothing property at the machine language interface. Appendix 9–A describes some examples of machine architectures that had this problem and the techniques that were used to add virtual memory to them.

A third example is the supervisor call (SVC). Chapter 5, on page 5–41, pointed out that the SVC instruction, which changes both the program counter and the processor mode bit (and in systems with virtual memory, other registers such as the page map address register), needs to be all-or-nothing, to assure that all (or none) of the intended registers change. Beyond that, the SVC invokes some complete kernel procedure. The designer would like to arrange that the entire call, (the combination of the SVC instruction and the operation of the kernel procedure itself) be an all-or-thing action. An all-or-nothing design allows the application programmer to view the kernel procedure as if it is an extension of the hardware. That goal is easier said than done, since the kernel procedure may detect some condition that prevents it from carrying out the intended action. Careful design of the kernel procedure is thus required.

Consider an SVC to a kernel READ procedure that delivers the next typed keystroke to the caller. The user may not have typed anything yet when the application program calls READ, so the the designer of READ must arrange to wait for the user to type something. By itself, this situation is not especially problematic, but it becomes more so when there is also a user-provided exception handler. Suppose, for example, a thread timer can expire during the call to READ and the user-provided exception handler is to decide whether or not the thread should continue to run a while longer. The scenario, then, is the user program calls READ, it is necessary to wait, and while waiting, the timer expires and control pases to the exception handler. Different systems choose one of three possibilities for the design of the READ procedure, the last one of which is not an all-or-nothing design:

> *1. An all-or-nothing design that implements the "nothing" option (blocking read)*: Seeing no available input, the kernel procedure first adjusts return pointers ("push the PC back") to make it appear that the application program called AWAIT just ahead of its call to the kernel READ procedure and then it transfers control to the kernel AWAIT entry point. When the user finally types something and AWAIT returns, the user's thread re-executes the original kernel call to READ, this time finding the typed input. With this design, if the timer exception occurs while waiting, when the exception handler investigates the current state of the thread it finds the answer "the application program is between instructions; its next instruction is a call to READ." This description is intelligible to a user-provided exception handler, and it allows that handler several options. One option is to continue the thread, meaning go ahead and execute the call to READ. If there is still no input, READ will again push the PC back and transfer control to AWAIT. Another option is for the handler to save this state description with a plan of restoring a future thread to this state at some later time.
>
> *2. An all-or-nothing design that implements the "all" option (non-blocking read)*: Seeing no available input, the kernel immediately returns to the application program with a zero-length result, expecting that the program will look for and properly handle this case. The program would probably test the length of the result and if zero, call AWAIT itself or it might find something else to do instead. As with the previous design, this design assures that at all times the user-provided timer exception handler will see a simple description of the current state of the thread—it is between two user program instructions. However, some care is needed to avoid a race between the call to AWAIT and the arrival of the next typed character.
>
> *3. A blocking read design that is neither "all" nor "nothing" and therefore not atomic*: The kernel READ procedure itself calls AWAIT, blocking the thread until the user types a character. Although this design seems conceptually simple, the description of the state of the thread from the point of view of the timer excpetioin handler is not simple. Rather than "between two user instructions", it is "waiting for something to happen in the middle of a user call to kernel procedure READ". The option of saving this state description for future use has been foreclosed. To start another thread with this state description, the exception handler would need to be able to request "start this thread just after the call to AWAIT in the middle of the kernel READ entry." But allowing that kind of request would compromise the modularity of the user-kernel interface. The user-provided exception handler could equally well make a request to restart the thread anywhere in the kernel, thus bypassing its gates and compromising its security.

The first and second designs correspond directly to the two options in the definition of an all-or-nothing action, and indeed some operating systems offer both options. In the first design the kernel program acts in a way that appears that the call had never taken place, while in the second design the kernel program runs to completion every time it is called. Both designs make the kernel procedure an all-or-nothing action, and both lead to a user-intelligible state description—the program is between two of its instructions—if an exception should happen while waiting.

One of the appeals of the client/server model introduced in chapter 4 is that it tends to force the all-or-nothing property out onto the design table. Because servers can fail independently of clients, it is necessary for the client to think through a plan for recovery from server failure, and a natural model to use is to make every action offered by a server all-or-nothing.

4. Coordinating concurrent threads

In chapter 5 we learned how to express opportunities for concurrency by creating threads, the goal of concurrency being to improve performance by running several things at the same time. Moreover, section A.1 of this chapter pointed out that interrupts can also create concurrency. Concurrent threads do not represent any special problem until their paths cross. The way that paths cross can always be described in terms of shared, writable data: concurrent threads happen to take an interest in the same piece of writable data at about the same time. It is not even necessary that the concurrent threads be running simultaneously; if one is stalled (perhaps because of an interrupt) in the middle of an action, a different, running thread can take an interest in the data that the stalled thread was, and will sometime again be, working with.

From the point of view of the programmer of an application, chapter 5 introduced two quite different kinds of concurrency coordination requirements: *sequence coordination* and *isolation*. Sequence coordination is a constraint of the type "Action *W* must happen before action *X*". For correctness, the first action must complete before the second action begins. For example, reading of typed characters from a keyboard must happen before running the program that presents those characters on a display. As a general rule, when writing a program one can anticipate the sequence coordination constraints, and the programmer knows the identity of the concurrent activities. Sequence coordination thus is usually explicitly programmed, using either special language constructs or shared variables such as the eventcounts of chapter 5.

In contrast, *isolation* is a more general constraint that several actions that concurrently operate on the same data should not interfere with one another. In chapter 5 we saw how isolation can be done with explicit locks and a thread manager that implements the procedures ACQUIRE and RELEASE. Chapter 5 showed some examples of isolation using locks, and emphasized that doing isolation correctly can be a tricky proposition. The thing that makes isolation harder than sequence coordination is that the programmer of an action that must be isolated does not necessarily know the identities of all the other actions, or of the threads that are running those actions. This lack of knowledge makes it difficult to accomplish isolation by explicit program steps, and thus the programmer needs an automatic, implicit mechanism that ensures isolation. Atomicity is a mechanism that provides such automatic, implicit isolation. We thus define isolation as an atomicity concept as follows:

Isolation atomicity

Two sequences of steps are *isolated* if their effect from the points of view of their invokers is the same as if the sequences occurred either *completely before* or *completely after* one another.

Brief version: *Do it all before or all after*.

In a specific case, one might say that "either the data must be Gronked before it is Munged, or else it must be Munged before it is Gronked." That way of putting it emphasizes that isolation atomicity always involves shared data. Correct coordination thus involves discipline in the way threads read and write shared data.

Let us consider that definition of isolation atomicity in the example of a banking application. We define a procedure named TRANSFER that debits one account and credits a second account, as follows (recall that the declaration keyword **reference** means that the argument is call-by-reference, so that TRANSFER can change the values of those arguments):

```
procedure TRANSFER (reference debit_account, reference credit_account, amount)
{
        debit_account ← debit_account - amount;
        credit_account ← credit_account + amount;
        return;
}
```

Despite their unitary appearance, program statements such as "*X* ← *X* + *Y*" are actually composite: they involve reading the values of *X* and *Y*, performing an addition, and then writing the result back into *X*. If a concurrent thread reads and changes the value of *X* between the read and the write done by this statement, that other thread may be surprised when this statement overwrites its change.

Suppose this procedure is applied to accounts *A* (initially containing $300) and *B* (initially containing $100) as in

```
        TRANSFER (A, B, $10);
```

We expect account *A*, the debit account, to end up with $290, and account *B*, the credit account, to end up with $110. Suppose, however, a second, concurrent thread is executing the statement

```
        TRANSFER (B, C, $25);
```

where account *C* starts with $175. When both threads complete their transfers, we expect *B* to end up with $85 and *C* with $200. Further, this expectation should be fulfilled no matter which of the two transfers happens first. But the variable *credit_account* in the first thread is bound to the same object (account *B*) as the variable *debit_account* in the second thread. The risk to correctness occurs if the two transfers happen at about the same time. To understand this risk, consider figure 9–2, which illustrates several possible time sequences of the READ and WRITE steps of the two threads with respect to variable *B*. With each time sequence the figure shows the history of values of the cell containing the balance of account *B*. If both steps 1–1 and 1–2 precede both steps 2–1 and 2–2, (or vice-versa) the two transfers will work as anticipated, and *B* ends up with $85. If, however, step 2–1 occurs after step 1–1, but before step 1–2, a mistake will occur: one of the two transfers will not affect account *B*, even though it should have. The first two cases illustrate histories of shared variable *B* in which the answers are the correct result; the remaining four cases illustrate four different sequences that lead to two incorrect values for *B*.

Thus our goal is to ensure that one of the first two time sequences actually occurs. One way to achieve this goal is that the two steps 1–1 and 1–2 should be atomic, and the two steps 2–1 and 2–2 should similarly be atomic. In the original program, the steps

debit_account ← *debit_account* - *amount*;

and

credit_account ← *credit_account* + *amount*;

should each be atomic. There should be no possibility that a concurrent thread that intends to change the value of the shared variable *debit_account* read its value between the READ and WRITE steps of this statement.

The notion that the first two sequences of figure 9–2 are correct and the other four are wrong is based on understanding this particular application. It would be better to have a more general concept of correctness that is independent of the application. Application independence is a modularity goal: we want to be able to make an argument for correctness of the isolation mechanism without getting into the question of whether or not the application using the mechanism is correct.

There is such a correctness concept: coordination among concurrent actions can be considered to be correct *if every result is guaranteed to be one that could have been obtained by some purely serial application* of those same actions.

The reasoning behind this concept of correctness involves several steps. Consider figure 9–3,which shows, abstractly, the effect of applying some action, whether atomic or not, to a system: the action changes the state of the system. Now, if we are sure that:

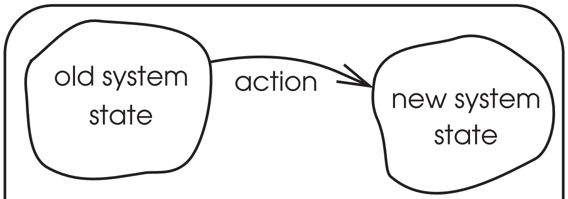


Figure 9–3: A single action takes a system from one proper state to another proper state.

1. the old state of the system was correct from the point of view of the application, and

2. the action, performing all by itself, correctly transforms any correct old state to a correct new state,

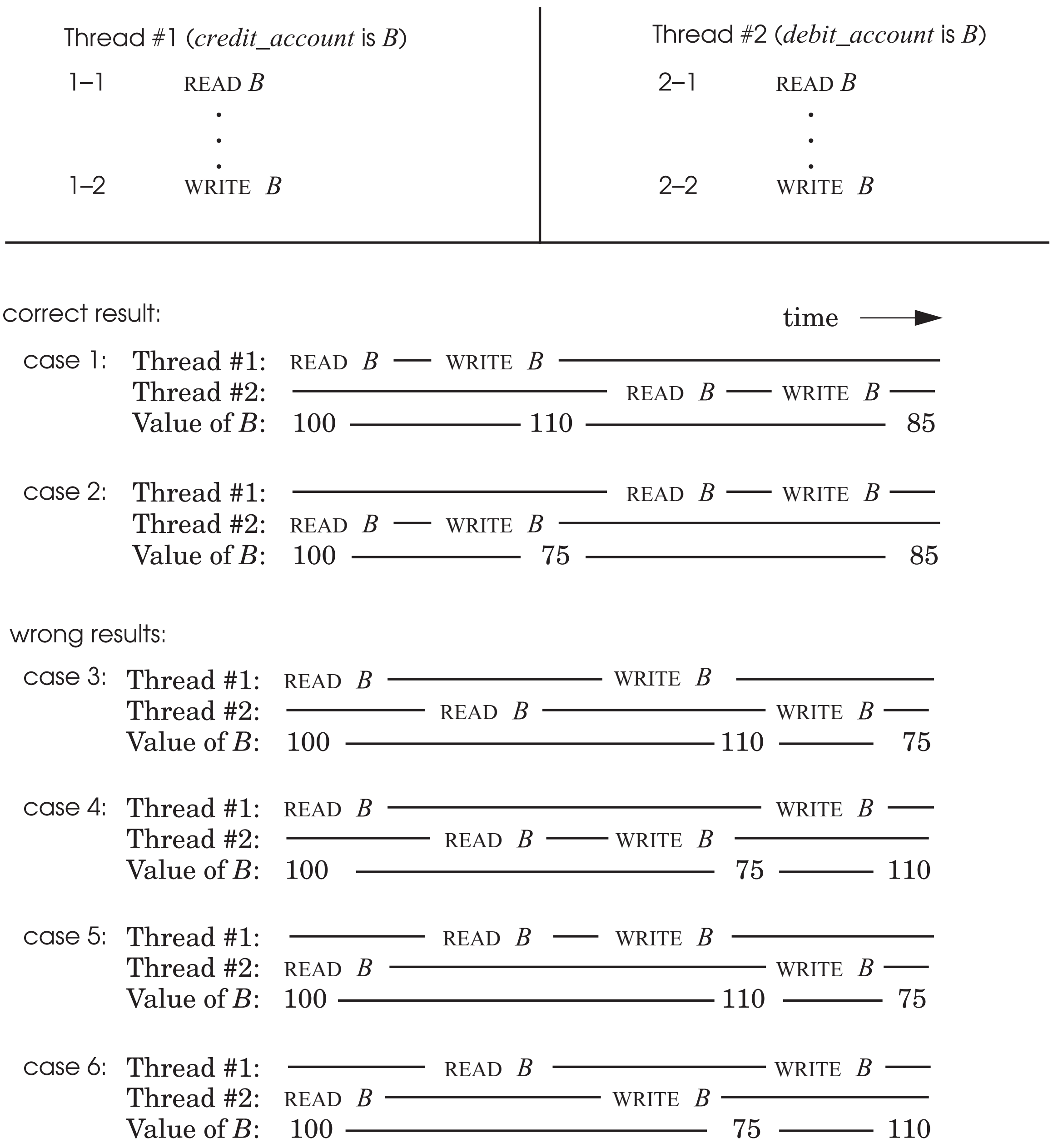

Thread #1 (credit_account is B)
1–1 READ B
1–2 WRITE B
Thread #2 (debit_account is B)
2–1 READ B
2–2 WRITE B
correct result:
time
case 1: Thread #1: READ B — WRITE B
Thread #2: READ B — WRITE B
Value of B: 100 110 85
case 2: Thread #1: READ B — WRITE B
Thread #2: READ B — WRITE B
Value of B: 100 75 85
wrong results:
case 3: Thread #1: READ B WRITE B
Thread #2: READ B WRITE B
Value of B: 100 110 75
case 4: Thread #1: READ B WRITE B
Thread #2: READ B WRITE B
Value of B: 100 75 110
case 5: Thread #1: READ B WRITE B
Thread #2: READ B WRITE B
Value of B: 100 110 75
case 6: Thread #1: READ B WRITE B
Thread #2: READ B WRITE B
Value of B: 100 75 110

Figure 9–2: Six possible histories of variable *B* if two threads that share *B* do not coordinate their concurrent activities.

then we can reason that the new state must also be correct. This line of reasoning holds for any application-dependent definition of "correct" and "correctly transform", so our reasoning method is independent of those definitions and thus of the application.

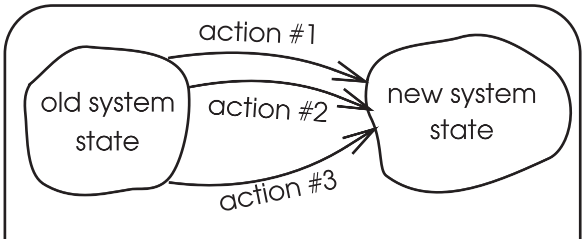


Figure 9–4: When several actions act concurrently, they together produce a new state. If the actions are isolated and the old state was proper, the new state will be proper.

The corresponding requirement when several actions act concurrently, as in figure 9–4, is that the resulting new state ought to be one of those that would have resulted from some serialization of the several actions, as in figure 9–5. This correctness criterion means that concurrent actions are correctly coordinated if their result is guaranteed to be one that would have been obtained by *some* purely serial application of those same actions. So long as the only coordination requirement is isolation atomicity, any serialization will do.

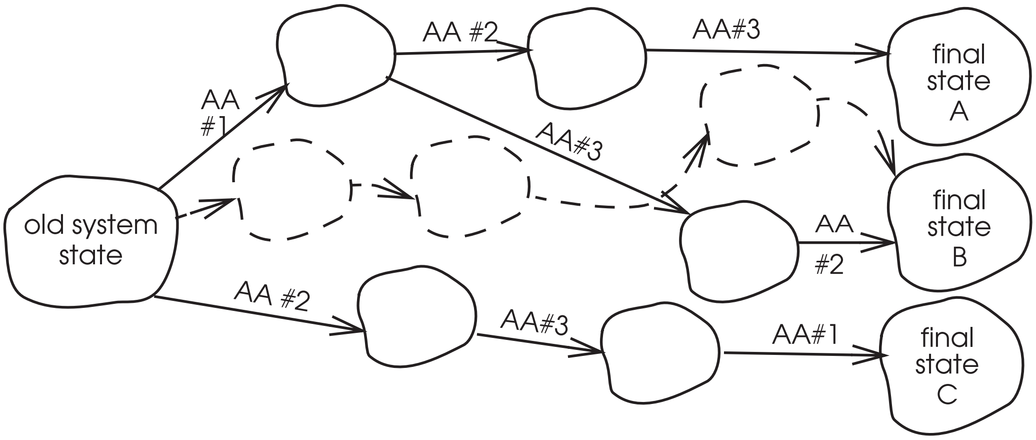


Figure 9–5: We insist that the final state be one that could have been reached by some serialization of the atomic actions, but we don't care which serialization. In addition, we do not need to insist that the intermediate states ever actually exist. The actual state trajectory could be that shown by the dotted lines, *but only if there is no way of observing the intermediate states from the outside*.

Moreover, we do not even need to insist that the system actually traverse the intermediate states along any particular path of figure 9–5—it may instead follow the dotted trajectory through intermediate states that are not by themselves proper, according to the application’s definition. As long as the intermediate states are not visible above the implementing layer, and the system is guaranteed to end up in one of the acceptable final states, we can declare the coordination to be proper, and therefore correct, because there exists a trajectory that leads to that state for which a propriety argument could have been applied to every step.

Since our definition of isolation atomicity is that each isolated action act as though it ran either completely before or completely after each other isolated action, isolation atomicity leads directly to this concept of correctness. Put another way, isolation atomicity has the effect of serializing the actions, so it follows that isolation atomicity guarantees correctness of coordination. A different way of expressing this idea is to say that when concurrent actions are correctly isolated, they are *serializable.* Thus in figure 9–2, the sequences of case 1 and case 2 could result from a serialized order, but the actions of cases 3 through 6 could not.

In some applications it is appropriate to use a correctness requirement that is stronger than serializability. For example, the designer of a banking system may want to avoid anachronisms by requiring *external time consistency*: if there is any external evidence (such as a printed receipt) that isolated action T_1 ended before isolated action T_2 began, the serialization order of T_1 and T_2 inside the system should be that T_1 precedes T_2. For another example of a stronger correctness requirement, a processor architect may require *sequential consistency*: when the processor concurrently performs multiple instructions from the same instruction stream, the result should be as if the instructions were executed in the original order specified by the programmer.

Returning to our example, a real funds-transfer application typically has several distinct isolation atomicity requirements. Consider the following auditing procedure; its purpose is to verify that the sum of the balances of all accounts is zero (accounts in debit status have negative balances):

```
procedure AUDIT()
{
        $sum \leftarrow 0$;
        for each $W \leftarrow$ in $bank.accounts$
                $sum \leftarrow sum + W.balance$;
        if ($sum \neq 0$) call for investigation;
        return;
}
```

Suppose that AUDIT is running in one thread at the same time that another thread is transferring money from account *A* to account *B*. If AUDIT examines account *A* before the transfer and account *B* after the transfer, it will count the transferred amount twice and thus will compute an incorrect answer. So the entire auditing procedure should occur either before or after any individual transfer: we want it to be an isolated action.

There is yet another isolation atomicity requirement: if AUDIT should run after the statement in TRANSFER

debit_account ← *debit_account* - *amount*;

but before the statement

credit_account ← *credit_account* + *amount*;

it will calculate a sum that does not include *amount*; we therefore conclude that the two balance updates taken together ought to be an isolated action. That is, the entire TRANSFER action should occur either completely before or completely after any AUDIT action; put another way, TRANSFER should be an isolated action.

5. *All-or-nothing and isolation atomicity*

We now have seen two forms of atomicity: all-or-nothing and isolation. However, they are not that different. The underlying goal common to both is to hide the composite nature of a multi-step action. With that insight, it becomes apparent that atomicity is really a unifying concept:

Atomicity

An action is atomic if there is no way for a higher layer to discover that its implementation is composite.

This description is really the fundamental definition of atomicity. From it, one can immediately draw two important consequences, corresponding to all-or-nothing atomicity and to isolation atomicity:

> *1.* From the point of view of a procedure that invokes an atomic action, the atomic action always appears either to complete as anticipated, or to do nothing. This consequence is the one that makes atomic actions useful in recovering from failures.
>
> *2.* From the point of view of a concurrent thread, an atomic action acts as though it occurs either *completely before* or *completely after* every other concurrent atomic action. This consequence is the one that makes atomic actions useful for coordinating concurrent threads.

These two consequences are not fundamentally different. They are simply two perspectives, the first from other modules within the thread that invokes the action, the second from other threads. Both points of view follow from the single idea that the composite nature of the action is not visible outside of the module that implements the action. Such hiding of internal structure is the essence of modularity, but atomicity is an exceptionally strong form of modularity. Atomicity hides not just the details of which steps form the atomic action, but the very fact that it is composite. There is a kinship between atomicity and other system-building techniques such as data abstraction and client/server organization. Data abstraction has the goal of hiding the composite nature of data; client/server organization has the goal of hiding the composite nature of major subsystems. Similarly, atomicity has the goal of hiding the composite nature of a multi-step procedure. All three are methods of enforcing industrial-strength modularity, and thereby of guaranteeing absence of unanticipated interactions among components of a complex system.

We have used phrases such as “from the point of view of the invoker” several times, suggesting that there may be another point of view from which internal structure is apparent. That other point of view is seen by the implementer of an atomic action, who is often painfully aware that an action is actually composite, and who must do extra work to provide the necessary assurances to the higher layer and to concurrent threads. Thus the interfaces between layers are an essential part of the definition of an atomic action, and they provide an opportunity for the implementation of an action to operate in any way that ends up providing atomicity.

There is one more aspect of hiding the composite nature of atomic actions: atomic actions can have benevolent side effects. A common example is an audit log, in which atomic actions that run into trouble record the nature of the detected failure and the recovery sequence for later analysis. One might think that when a failure leads to backing out, the audit log should be rolled back, too; but rolling it back would defeat its purpose—the whole point of an audit log is to remember details about the failure. The important point is that the audit log is normally a private record of the layer that implemented the atomic action; in the normal course of operation it is not visible above that layer, so there is no requirement to roll it back. (A separate atomicity requirement is to ensure that the log entry that describes a failure is complete and not lost in the ensuing recovery.)

Another example of a benevolent side effect is performance optimization. For example, in a high-performance data management system, when an upper layer atomic action asks the data management system to insert a new record into a file, the data management system may decide as a performance optimization that now is the time to rearrange the file into a better physical order. If the atomic action fails and aborts, it need ensure only that the newly-inserted record be removed; the file does not need to be restored to its older, less efficient, storage arrangement. Similarly, a lower-layer cache that now contains a variable touched by the atomic action does not need to be cleared and a garbage collection of heap storage does not need to be undone. Such side effects are not a problem, as long as they are hidden from the higher-layer client of the atomic action except perhaps in the speed with which later actions are carried out, or across an interface that is intended to report performance measures or failures.

B. All-or-nothing atomicity I: Concepts

Section A of this chapter defined the goals of all-or-nothing atomicity and isolation atomicity, and provided a conceptual framework that at least in principle allows a designer to decide whether or not some proposed algorithm correctly isolates concurrent activities. However, it did not provide any examples of actual implementations of either goal. This section of the chapter, together with the next one, describe some widely applicable techniques of systematically implementing all-or-nothing atomicity. Later sections of the chapter will do the same for isolation atomicity.

Many of the examples employ the technique introduced in chapter 5 called *bootstrapping*, a method that resembles inductive proof. To review, bootstrapping means to first look for a systematic way to reduce a general problem to some much-narrowed particular version of that same problem. Then, solve the narrow problem using some specialized method that might work only for that case, because it takes advantage of the specific situation. The general solution then consists of two parts: a special-case technique plus a method that systematically reduces the general problem to the special case. Recall that chapter 5 tackled the general problem of isolating arbitrary sequences of code by implementing a procedure named ACQUIRE that itself required isolation of two or three lines of code where it reads and then sets a lock value. It then implemented that isolation with the help of a special hardware feature that directly isolates the read and set sequence, and it also exhibited a software implementation (in sidebar 5–2 on page 5–30) that does not require special hardware. This chapter uses bootstrapping several times. The first example starts with the special case and then introduces a way to reduce the general problem to that special case.

1. *Achieving all-or-nothing atomicity:* ALL_OR_NOTHING_PUT

The first example is of a scheme that does an all-or-nothing update of a single disk sector. The problem to be solved is that if a system crashes in the middle of a disk write (for example, the operating system encounters a bug or the power fails), the sector that was being written at the instant of the failure may contain an unusable muddle of old and new data. The goal is to create an all-or-nothing PUT with the property that when GET later reads the sector, it always returns either the old or the new data, but never a muddled mixture.

To make the implementation precise, we develop a disk fault tolerance model that is a slight variation of the one introduced in chapter 8, taking as an example application a calendar management program for a personal computer. The user is hoping that, if the system fails while adding a new event to the calendar, when the system later restarts the calendar will be safely intact. Whether or not the new event ended up in the calendar is less important than that the calendar not be damaged by inopportune timing of the system failure. This system comprises a human user, a display, a processor, some volatile memory, a magnetic disk, an operating system, and the calendar manager program. We model this system in several parts:

Overall system fault tolerance model

- error-free operation: All work goes according to expectations. The user initiates actions such as adding events to the calendar and the system confirms the actions by displaying messages to the user.

- tolerated error: The user who has initiated an action notices that the system failed before it confirmed completion of the action and, when the system is operating again, checks to see whether or not it actually performed that action.

- untolerated error: The system fails without the user noticing, so the user does not realize that he or she should check or retry an action that the system may not have completed.

The tolerated error specification means that, to the extent possible, the entire system is fail-fast: if something goes wrong during an update, the system stops before taking any more requests, and the user realizes that the system has stopped. One would ordinarily design a system such as this one to minimize the chance of the untolerated error, for example by requiring supervision by a human user. The human user then is in a position to realize (perhaps from lack of response) that something has gone wrong. After the system restarts, the user knows to inquire whether or not the action completed. This design strategy should be familiar from our study of best effort networks in chapter 7. The lower layer (the computer system) is providing a best effort implementation. A higher layer (the human user) supervises and, when necessary, retries. For example, suppose that the human user adds an appointment to the calendar but just as he or she clicks “save” the system crashes. The user doesn’t know whether or not the addition actually succeeded, so when the system comes up again the first thing to do is open up the calendar to find out what happened.

Processor, memory, and operating system fault tolerance model

This part of the model just specifies more precisely the intended fail-fast properties of the hardware and operating system:

- error-free operation: The processor, memory, and operating system all follow their specifications.

- detected error: Something fails in the hardware or operating system. The system is fail-fast: the hardware or operating system detects the failure and restarts from a clean slate *before* initiating any further PUTs to the disk.

- untolerated error: Something fails in the hardware or operating system. The processor muddles along and PUTs corrupted data to the disk before detecting the failure.

The primary goal of the processor/memory/operating-system part of the model is to detect failures and stop running before any corrupted data is written to the disk storage system. The importance of detecting failure before the next disk write lies in error containment: if the goal is met, the designer can assume that the only values potentially in error must be in processor registers and volatile memory, and the data on the disk should be safe, with the exception described in chapter 8 on page 8–51: if there was a PUT to the disk in progress at the time of the crash, the failing system may have corrupted the disk buffer in volatile memory, and consequently corrupted the disk sector that was being written.

The recovery procedure can thus depend on the disk storage system to contain only uncorrupted information, or at most one corrupted disk sector. In fact, after restart the disk will contain the *only* information. “Restarts from a clean slate” means that the system discards all state held in volatile memory. This step brings the system to the same state as if a power failure had occurred, so a single recovery procedure will be able to handle both system crashes and power failures. Discarding volatile memory also means that all currently active threads vanish, so everything that was going on comes to an abrupt halt and will have to be restarted.

Disk storage system fault tolerance model

Implementing all-or-nothing atomicity involves some steps that resemble the decay masking of MORE_DURABLE_PUT/GET in chapter 8—in particular, the algorithm will write multiple copies of data. To clarify how the all-or-nothing mechanism works, we temporarily back up to CAREFUL_PUT/GET (see page 8–54), which masks soft disk errors but not hard disk errors or disk decay. To simplify further, we pretend for the moment that a disk never decays and that it has no hard errors. (Since this perfect-disk assumption is obviously unrealistic, we shall reverse it in appendix 9–B, which describes an algorithm for all-or-nothing atomicity despite disk decay and hard errors.)

With the perfect-disk assumption, only one thing can go wrong: a system crash at just the wrong time. The fault tolerance model for this simplified careful disk system then becomes:

- error-free operation: CAREFUL_GET returns the result of the most recent call to CAREFUL_PUT at *sector_number* on *track*, with *status* = OK.

- detectable error: The operating system crashes during a CAREFUL_PUT and corrupts the disk buffer in volatile storage, and CAREFUL_PUT writes corrupted data on one sector of the disk.

We classify the error as “detectable” because we assume that the application has included with the data an end-to-end checksum, calculated before calling CAREFUL_PUT and thus before the system crash could have corrupted the data.

The change in this revision of the careful storage layer is that when a system crash occurs, one sector on the disk may be corrupted, but the client of the interface is confident that (1) that sector is the only one that may be corrupted and (2) if it has been corrupted, any later reader of that sector will detect the problem. Between the processor model and the storage system model, all anticipated failures now lead to the same situation: the system detects the failure, resets all processor registers and volatile memory, forgets all active threads, and restarts. No more than one disk sector is corrupted.

Our problem is now reduced to providing the all-or-nothing property: the goal is to create *all-or-nothing disk storage*, which guarantees either to change the data on a sector completely and correctly or else appear to future readers not to have touched it at all. Here is one simple, but somewhat inefficient, scheme that makes use of virtualization: assign, for

```
procedure ALMOST_ALL_OR_NOTHING_PUT (data, all_or_nothing_sector) {
        CAREFUL_PUT (data, all_or_nothing_sector.S1);
        CAREFUL_PUT (data, all_or_nothing_sector.S2);          // Commit point.
        CAREFUL_PUT (data, all_or_nothing_sector.S3);
}

procedure ALL_OR_NOTHING_GET (reference data, all_or_nothing_sector) {
        CAREFUL_GET (data1, all_or_nothing_sector.S1);
        CAREFUL_GET (data2, all_or_nothing_sector.S2);
        CAREFUL_GET (data3, all_or_nothing_sector.S3);
        if data1 = data2 then data ← data1;                    // Return new value.
        else data ← data3;                                     // Return old value.
}
```

Figure 9–6: Algorithms for ALMOST_ALL_OR_NOTHING_PUT and ALL_OR_NOTHING_GET.

each data sector that is to have the all-or-nothing property, three physical disk sectors, identified as *S1*, *S2*, and *S3.* The three physical sectors taken together are a virtual "all-or-nothing sector". At each place in the system where this disk sector was previously used, replace it with the all-or-nothing sector, identified by the triple {*S1*, *S2*, *S3*}. We start with an almost correct all-or-nothing implementation named ALMOST_ALL_OR_NOTHING_PUT, find a bug in it, and then fix the bug, finally creating a correct ALL_OR_NOTHING_PUT.

When asked to write data, ALMOST_ALL_OR_NOTHING_PUT writes it three times, on *S1*, *S2*, and *S3,* in that order, each time waiting until the previous write finishes, so that if the system crashes only one of the three sectors will be affected. To read data, ALL_OR_NOTHING_GET reads all three sectors and compares their contents. If the contents of *S1* and *S2* are identical, ALL_OR_NOTHING_GET returns that value as the value of the all-or-nothing sector. If *S1* and *S2* differ, ALL_OR_NOTHING_GET returns the contents of *S3* as the value of the all-or-nothing sector. Figure 9–6 shows this almost correct pseudocode.

Let's explore how this implementation behaves on a system crash. Suppose that at some previous time a record has been correctly stored in an all-or-nothing sector, and someone now updates it by calling ALL_OR_NOTHING_PUT. The goal is that even if a failure occurs in the middle of the update, a later reader can always be assured of getting some complete, consistent version of the record by invoking ALL_OR_NOTHING_GET.

Suppose that ALMOST_ALL_OR_NOTHING_PUT were interrupted by a system crash some time before it finishes writing sector *S2*, and thus corrupts either *S1* or *S2*. In that case, when ALL_OR_NOTHING_GET reads sectors *S1* and *S2*, they will have different values, and it is not clear which one to trust. Because the system is fail-fast, sector *S3* would not yet have been touched by ALMOST_ALL_OR_NOTHING_PUT, so it still contains the previous value. Returning the value found in *S3* thus has the desired effect of ALMOST_ALL_OR_NOTHING_PUT having done nothing.

Now, suppose that ALMOST_ALL_OR_NOTHING_PUT were interrupted by a system crash some time after successfully writing sector *S2*. In that case, the crash may have corrupted *S3*, but *S1* and *S2* both contain the newly updated value. ALL_OR_NOTHING_GET returns the value of *S1*, thus providing the desired effect of ALMOST_ALL_OR_NOTHING_PUT having completed its job.

```
1    procedure ALL_OR_NOTHING_PUT (data, all_or_nothing_sector) {
2        CHECK_AND_REPAIR (all_or_nothing_sector);
3        ALMOST_ALL_OR_NOTHING_PUT (data, all_or_nothing_sector);
4    }

5    procedure CHECK_AND_REPAIR (all_or_nothing_sector) {        // Make sure all three copies
match.
6        CAREFUL_GET (data1, all_or_nothing_sector.S1);
7        CAREFUL_GET (data2, all_or_nothing_sector.S2);
8        CAREFUL_GET (data3, all_or_nothing_sector.S3);
9        if (data1 = data2) and (data2 = data3) return;          // State 1 or 7, nothing to do
10       if (data1 = data2)
11           {CAREFUL_PUT (data1, all_or_nothing_sector.S3); return;}    // State 5 or 6.
12       if (data2 = data3)
13           {CAREFUL_PUT (data2, all_or_nothing_sector.S1); return;}    // State 2 or 3.
14       CAREFUL_PUT (data1, all_or_nothing_sector.S2);          // State 4, change to state 5.
15       CAREFUL_PUT (data1, all_or_nothing_sector.S3);          // State 5, change to state 6.
16       return;
17   }
```

Figure 9–7: Algorithms for ALL_OR_NOTHING_PUT and CHECK_AND_REPAIR.

So what's wrong with this design? ALMOST_ALL_OR_NOTHING_PUT assumes that all three copies are identical when it starts. But a previous failure can violate that assumption. Suppose that ALMOST_ALL_OR_NOTHING_PUT is interrupted while writing *S3*. The next thread to call ALL_OR_NOTHING_GET finds *data1* = *data2*, so it uses *data1*, as expected. The new thread then calls ALMOST_ALL_OR_NOTHING_PUT, but is interrupted while writing *S2*. Now, *S1* doesn't equal *S2*, so the next call to ALMOST_ALL_OR_NOTHING_PUT returns the damaged *S3*.

The fix for this bug is for ALL_OR_NOTHING_PUT to guarantee that the three sectors be identical before updating. It can provide this guarantee by invoking a procedure named CHECK_AND_REPAIR as in figure Figure 9–7. CHECK_AND_REPAIR simply compares the three copies and, if they are not identical, it forces them to be identical. To see how this works, assume that someone calls ALL_OR_NOTHING_PUT at a time when all three of the copies do contain identical values, which we designate as "old". Because ALL_OR_NOTHING_PUT writes "new" values into *S1*, *S2*, and *S3* one at a time and in order, even if there is a crash, at the next call to ALL_OR_NOTHING_PUT there are only seven possible data states for CHECK_AND_REPAIR to consider:

data state:	1	2	3	4	5	6	7
sector *S1*	old	bad	new	new	new	new	new
sector *S2*	old	old	old	bad	new	new	new
sector *S3*	old	old	old	old	old	bad	new

The way to read this table is as follows: if all three sectors *S1*, *S2*, and *S3* contain the "old" value, the data is in state 1. Now, if CHECK_AND_REPAIR discovers that all three copies are identical (line 9 in figure 9–7), the data is in state 1 or state 7 so CHECK_AND_REPAIR simply returns. Failing that test, if the copies in sectors *S1* and *S2* are identical (line 10), the data must be in state 5 or state 6, so CHECK_AND_REPAIR forces sector *S3* to match and returns (line 11). If the copies in sectors *S2* and *S3* are identical the data must be in state 2 or state 3 (line 12), so CHECK_AND_REPAIR forces sector *S1* to match and returns (line 13). The only remaining possibility is that the data is in state 4, in which case sector *S2* is surely bad, but sector *S1* contains an old value and sector *S3* contains a new one. The choice of which to use is arbitrary; as shown the procedure copies the new value in sector *S1* to both sectors *S2* and *S3*.

What if a failure occurs while running CHECK_AND_REPAIR? That procedure systematically drives the state either forward from state 4 toward state 7, or backward from state 3 toward state 1. If CHECK_AND_REPAIR is itself interrupted by another system crash, rerunning it will continue from the point at which the previous attempt left off.

We can make several observations about the algorithm implemented by ALL_OR_NOTHING_GET and ALL_OR_NOTHING_PUT:

1. This all-or-nothing atomicity algorithm assumes that only one thread at a time tries to execute either ALL_OR_NOTHING_GET or ALL_OR_NOTHING_PUT. This algorithm implements all-or-nothing atomicity but not isolation atomicity.

2. CHECK_AND_REPAIR is *idempotent*. That means that a thread can start the procedure, execute any number of its steps, be interrupted by a crash, and go back to the beginning again any number of times with the same ultimate result, as far as a later call to ALL_OR_NOTHING_GET is concerned.

3. The completion of the CAREFUL_PUT on line *3* of ALMOST_ALL_OR_NOTHING_PUT, marked "commit point," exposes the new data to future ALL_OR_NOTHING_GET actions. Until that step begins execution, a call to ALL_OR_NOTHING_GET sees the old data. After line *3* completes, a call to ALL_OR_NOTHING_GET sees the new data.

4. Although the algorithm writes three replicas of the data, the primary reason for the replicas is not to provide durability as described in section E of chapter 8. Instead, the reason for writing three replicas, one at a time and in a particular order, is to ensure observance at all times and under all failure scenarios of the *golden rule of atomicity*, which is the subject of the next section.

There are several ways of implementing all-or-nothing disk sectors. Near the end of chapter 8 we introduced a fault tolerance model for decay events that did not mask system crashes, and applied the technique known as RAID to mask decay to produce durable storage. Here we started with a slightly different fault tolerance model that omits decay, and we devised techniques to mask system crashes and produce all-or-nothing storage. What we really should do is start with a fault tolerance model that considers both system crashes and decay, and devise storage that is both all-or-nothing and durable. Such a model, devised by Xerox Corporation researchers Butler Lampson and Howard Sturgis, is the subject of appendix 9–B, together with the more elaborate recovery algorithms it requires. That model has the additional feature that it needs only two physical sectors for each all-or-nothing sector.

2. *Systematically achieving all-or-nothing atomicity: commit and the golden rule*

The example of ALL_OR_NOTHING_PUT and ALL_OR_NOTHING_GET demonstrates an interesting special case of all-or-nothing atomicity, but it offers little guidance on how to systematically create a more general all-or-nothing action. From the example, our calendar program now has a tool that allows writing individual sectors with the all-or-nothing property, but that is not the same as safely adding an event to a calendar, since adding an event probably requires rearranging a data structure, which in turn may involve writing more than one disk sector. We could do a series of ALL_OR_NOTHING_PUTs to the several sectors, and be assured that each sector is itself written in an all-or-nothing fashion, but a crash that occurs after writing one and before writing the next would leave the overall calendar addition in a partly-done state. To make the entire calendar addition action all-or-nothing we need a generalization.

Ideally, one might like to be able to take any arbitrary sequence of instructions in a program, surround that sequence with some sort of **begin** and **end** statements as in figure 9–8, and expect that the language compilers and operating system will perform some magic that makes the surrounded sequence into an all-or-nothing action. Unfortunately, no one knows how to do that. But we can come close, if the programmer is willing to make a modest concession to the requirements of all-or-nothing atomicity. This concession is expressed in the form of a discipline on the constituent steps of the all-or-nothing action.

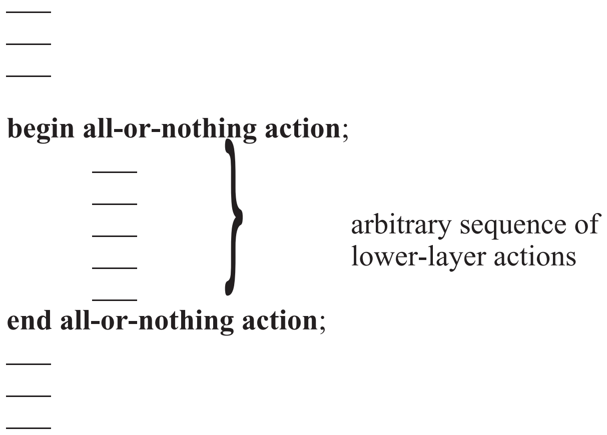

begin all-or-nothing action;
arbitrary sequence of lower-layer actions
end all-or-nothing action;

Figure 9–8: Imaginary semantics for painless programming of all-or-nothing actions.

The discipline starts by identifying some single step of the sequence as the *commit point*. The all-or-nothing action is thus divided into two phases, a *pre-commit phase* and a *post-commit phase*, as suggested by figure 9–9. During the pre-commit phase, the disciplining rule of design is that no matter what happens, it must be possible to back out of this all-or-nothing action in a way that leaves no trace. During the post-commit phase the disciplining rule of design is that no matter what happens, the action must run to the end successfully. Thus an all-or-nothing action can have only two outcomes. If the all-or-nothing action starts

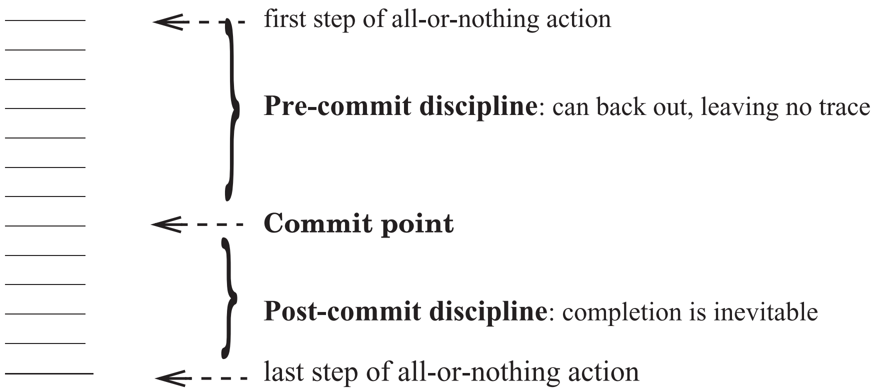

first step of all-or-nothing action
Pre-commit discipline: can back out, leaving no trace
Commit point
Post-commit discipline: completion is inevitable
last step of all-or-nothing action

Figure 9–9: The commit point of an all-or-nothing action.

and then, without reaching the commit point, backs out, we say that it *aborts*. If the all-or-nothing action passes the commit point, we say that it *commits*.

We can make several observations about the restrictions of the pre-commit phase. The pre-commit phase must identify all the resources needed to complete the all-or-nothing action, and establish their availability. The names of data should be bound, permissions should be checked, the pages to be read or written should be in memory, removable media should be mounted, stack space must be allocated, etc. In other words, all the steps needed to anticipate the severe run-to-the-end-without-faltering requirement of the post-commit phase should be completed during the pre-commit phase. In addition, the pre-commit phase must maintain the ability to abort at any instant. Any changes that the pre-commit phase makes to the state of the system must be undoable in case this all-or-nothing action aborts. Usually, this requirement means that shared resources, once reserved, cannot be released until the commit point is passed. The reason is that if an all-or-nothing action releases a shared resource, some other, concurrent thread may capture that resource. If the resource is needed in order to undo some effect of the all-or-nothing action, releasing the resource is tantamount to abandoning the ability to abort. Finally, the reversibility requirement means that the all-or-nothing action should not do anything externally visible, for example printing a check or firing a missile, prior to the commit point. (It is possible, though more complicated, to be slightly less restrictive. Sidebar 9–3 explores that possibility.)

In contrast, the post-commit phase can expose results, it can release reserved resources that are no longer needed, and it can perform externally visible actions such as printing a

Sidebar 9–3: Cascaded aborts

(Temporary) *sweeping simplification*: In this initial discussion of commit points, we are intentionally avoiding a more complex and harder-to-design possibility. Some systems allow other, concurrent activities to see pending results, and they may even allow externally visible actions before commit. Those systems must therefore be prepared to track down and abort those concurrent activities (this tracking down is called *cascaded abort*) or perform *compensating* external actions (e.g., send a letter requesting return of the check or apologizing for the missile firing). The discussion of layers and multiple sites in chapter 10 introduces a simple version of cascaded abort.

check, opening a cash drawer, or drilling a hole. But it cannot try to acquire additional resources, because an attempt to acquire might fail, and the post-commit phase is not permitted the luxury of failure. The post-commit phase must confine itself to finishing just the activities that were planned during the pre-commit phase.

It might appear that if a system fails before the post-commit phase completes, all hope is lost, so the only way to assure all-or-nothing atomicity is to always make the commit step the last step of the all-or-nothing action. Often, that is the simplest way to assure all-or-nothing atomicity, but the requirement is not actually that stringent. An important feature of the post-commit phase is that it is hidden inside the layer that implements the all-or-nothing action, so a scheme that assures that the post-commit phase completes *after* a system failure is acceptable, so long as this delay is hidden from the invoking layer. Some all-or-nothing atomicity schemes thus involve a guarantee that a cleanup procedure will be invoked following every system failure, or as a prelude to the next use of the data, before anyone in a higher layer gets a chance to discover that anything went wrong. This idea should sound familiar: the implementation of ALL_OR_NOTHING_PUT in figure 9–7 used this approach, by always running the cleanup procedure named CHECK_AND_REPAIR before updating the data.

A popular technique for achieving all-or-nothing atomicity is called the *shadow copy*. It is used by text editors, compilers, calendar management programs, and other programs that modify existing files, to ensure that following a system failure the user does not end up with data that is damaged or that contains only some of the intended changes:

- Pre-commit: Create a complete duplicate working copy of the file that is to be modified. Then, make all changes to the working copy.
- Commit point and post-commit: Carefully exchange the working copy with the original. Typically this step is bootstrapped, using a lower-layer RENAME entry point of the file system that provides certain atomic-like guarantees such as the ones described for the Unix version of RENAME on page 2–68. Release the space that was occupied by the original.

The ALL_OR_NOTHING_PUT algorithm of figure 9–7 can be seen as a particular example of the shadow copy strategy, which itself is a particular example of the general pre-commit/post-commit discipline. The commit point occurs at the instant when the new value of *S2* is successfully written to the disk. During the pre-commit phase, while ALL_OR_NOTHING_PUT is checking over the three sectors and writing the shadow copy *S1*, a crash will leave no trace of that activity (that is, no trace that can be discovered by a later caller of ALL_OR_NOTHING_GET). The post-commit phase of ALL_OR_NOTHING_PUT consists of writing *S3*.

From these examples we can extract an important design principle:

The golden rule of atomicity

Never modify the only copy!

In order for a composite action to be all-or-nothing, there must be some way of reversing the effect of each of its pre-commit phase component actions, so that if the action does not commit it is possible to back out. As we continue to explore implementations of all-or-nothing atomicity, we shall notice that correct implementations always reduce at the end to making a shadow copy. The reason is that structure ensures that the implementation follows the golden rule.

3. Systematically achieving all-or-nothing atomicity: version histories

This subsection develops a scheme to provide all-or-nothing atomicity in the general case of a program that modifies arbitrary data structures. It will be easy to see why the scheme is correct, but the mechanics can interfere with performance. Section C of this chapter then introduces a variation on the scheme that requires more thought to see why it is correct, but that allows higher-performance implementations. As before, we concentrate for the moment on all-or-nothing atomicity. While some aspects of isolation atomicity will also emerge, we leave a systematic treatment of that topic for discussion in sections D and E of this chapter. Thus the model to keep in mind in this section is that only a single thread is running. If the system crashes, after a restart the original thread is gone—recall from chapter 8 the *sweeping simplification* that threads are included in the volatile state that is lost on a crash and only durable state survives. After the crash, a new, different thread comes along and attempts to look at the data. The goal is that the new thread should always find that the all-or-nothing action that was in progress at the time of the crash either never started or completed successfully.

In looking at the general case, a fundamental difficulty emerges: random-access memory and disk usually appear to the programmer as a set of named, shared, and rewritable storage cells, called *cell storage*. Cell storage has semantics that are actually quite hard to make all-or-nothing, because the act of storing destroys old data, thus potentially violating the golden rule of atomicity. If the all-or-nothing action later aborts, the old value is irretrievably gone; at best it can only be reconstructed from information kept elsewhere. In addition, storing data reveals it to the view of later threads, whether or not the all-or-nothing action that stored the value reached its commit point. If the all-or-nothing action happens to have exactly one output value, then writing that value into cell storage can be the mechanism of committing, and there is no problem. But if the result is supposed to consist of several output values, all of which should be exposed simultaneously, it is harder to see how to construct the all-or-nothing action. Once the first output value is stored, the computation of the remaining outputs has to be successful; there is no going back. If the system fails and we have not been very careful, a later thread may see some old and some new values.

These limitations of cell storage did not plague the shopkeepers of Padua, who in the 14th century invented double-entry bookkeeping. Their storage medium was leaves of paper in bound books and they made new entries with quill pens. They never erased or even crossed out entries that were in error; when they made a mistake they made another entry that reversed the mistake, thus leaving a complete history of their actions, errors, and corrections in the book. It wasn't until the 1950's, when programmers began to automate bookkeeping systems, that the notion of overwriting data emerged. Up until that time, if a bookkeeper collapsed and died while making an entry, it was always possible for someone else to seamlessly take over the books. This observation about the robustness of paper systems suggests that there is a form of the golden rule of atomicity that might allow one to be systematic: never erase anything.

Examining the shadow copy technique used by the text editor provides a second useful idea. The essence of the mechanism that allows a text editor to make several changes to a file,

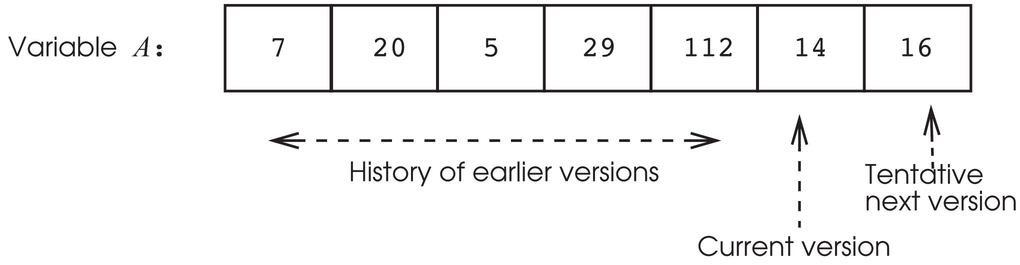


Figure 9–10: Version history of a variable in journal storage.

yet not reveal any of the changes until it is ready, is this: the only way another prospective reader of a file can reach it is by name. Until commit time the editor works on a copy of the file that is either not yet named or has a unique name not known outside the thread, so the modified copy is effectively invisible. Renaming the new version is the step that makes the entire set of updates simultaneously visible to later readers.

These two observations suggest that all-or-nothing actions would be better served by a model of storage that behaves differently from cell storage: instead of a model in which a store operation overwrites old data, we instead create a new, tentative version of the data, such that the tentative version remains invisible to any reader outside this all-or-nothing action until the action commits. We can provide such semantics, even though we start with traditional cell memory, by interposing a layer between the cell storage and the program that reads and writes data. This layer implements what is known as *journal storage*. The basic idea of journal storage is straightforward: we associate with every named variable not a single cell, but a list of cells in non-volatile storage; the values in the list represent the history of the variable. Figure 9–10 illustrates. Whenever any action proposes to write a new value into the variable, the journal storage manager appends the prospective new value to the end of the list. Clearly this approach, being history-preserving, offers some hope of being helpful, because if an all-or-nothing action aborts, one can imagine a systematic way to locate and discard all of the new versions it wrote. Moreover, we can tell the journal storage manager to expect to receive tentative values, but to ignore them unless the all-or-nothing action that created them commits. The basic mechanism to accomplish such an expectation is quite simple; the journal storage manager should make a note, next to each new version, of the identity of the all-or-nothing action that created it. Then, at any later time, it can discover the status of the tentative version by inquiring whether or not the all-or-nothing action ever committed.

Figure 9–11 illustrates the overall structure of such a journal storage system, implemented as a layer that hides a cell storage system. (To reduce clutter, this journal storage system omits calls to create new and delete old variables.) In this particular model, we assign to the journal storage manager most of the job of providing tools for programming all-or-nothing actions. Thus the implementer of a prospective all-or-nothing action should begin that action by invoking the journal storage manager entry NEW_ACTION, and later complete the action by invoking either COMMIT or ABORT. If, in addition, actions perform all reads and writes of data by invoking the journal storage manager's READ_CURRENT_VALUE and

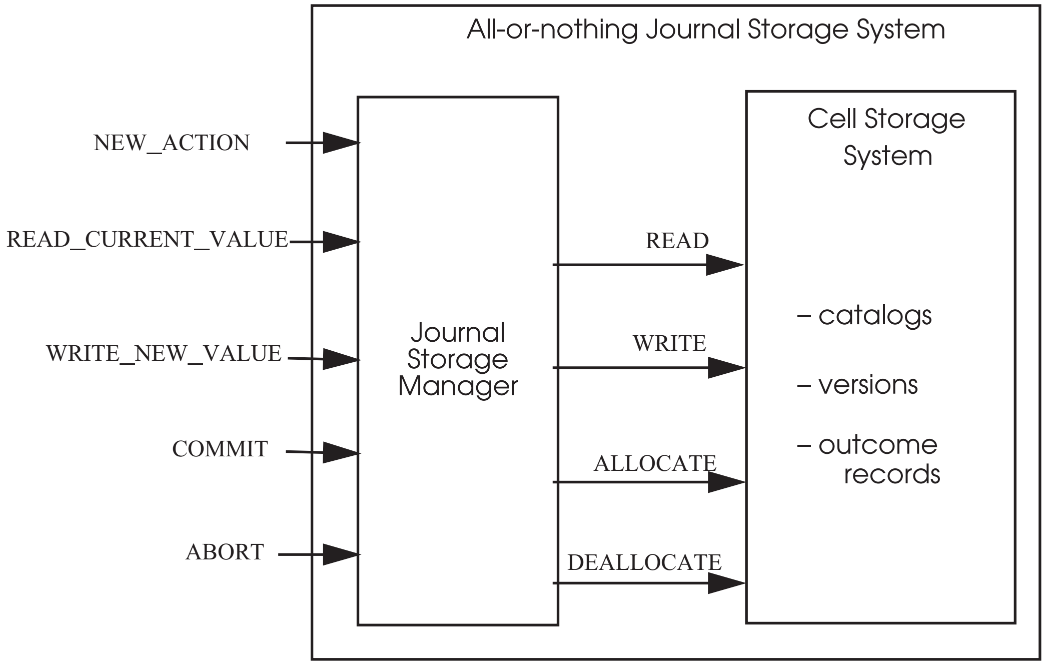


Figure 9–11: Interface to and internal organization of an all-or-nothing storage system based on version histories and journal storage.

WRITE_NEW_VALUE entries, our hope is that the result will automatically be all-or-nothing with no further concern of the implementer.

How could this automatic all-or-nothing atomicity work? The first step is that the journal storage manager, when called at NEW_ACTION, should assign a nonce identifier to the prospective all-or-nothing action, and create, in non-volatile cell storage, a record of this new identifier and the state of the new all-or-nothing action. This record is called an *outcome record*; it begins its existence in the state PENDING; depending on the outcome it should eventually move to one of the states COMMITTED or ABORTED, as suggested by figure 9–12. No other state transitions are possible, except to discard the outcome record once there is no further interest in its state. Figure 9–13 illustrates implementations of the three procedures NEW_ACTION, COMMIT, and ABORT.

When an all-or-nothing action calls the journal storage manager to write a new version of some data object, that action supplies the identifier of the data object, a tentative new value for the new version, and the identifier of the all-or-nothing action. The journal storage manager calls on the lower-level storage management system to allocate in non-volatile cell storage enough space to contain the new version; it places in the newly allocated cell storage the new data value and the identifier of the all-or-nothing action. Thus the journal storage manager creates a version history as illustrated in figure 9–14. Now, when someone proposes to read a data value by calling READ_CURRENT_VALUE, the journal storage manager can review the version history, starting with the latest version and return the value in the most recent committed version. By inspecting the outcome records, the journal storage manager can

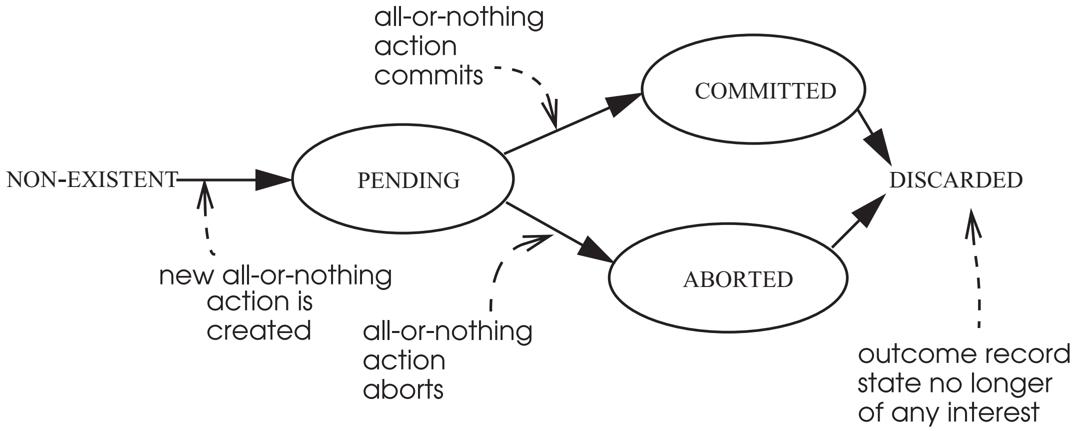

all-or-nothing action commits
COMMITTED
NON-EXISTENT
PENDING
DISCARDED
ABORTED
new all-or-nothing action is created
all-or-nothing action aborts
outcome record state no longer of any interest

Figure 9–12: The allowed state transitions of an outcome record.

```
procedure NEW_ACTION ()
{
        id ← NEW_OUTCOME_RECORD ();
        id.outcome_record.state ← PENDING;
        return id;
}

procedure COMMIT (reference id)
{
        id.outcome_record.state ← COMMITTED;
}

procedure ABORT (reference id)
{
        id.outcome_record.state ← ABORTED;
}
```

Figure 9–13: The procedures NEW_ACTION, COMMIT, and ABORT.

ignore those versions that were written by all-or-nothing actions that aborted or that never committed.

The procedures READ_CURRENT_VALUE and WRITE_NEW_VALUE thus follow the algorithms of figure 9–15. The important property of this pair of algorithms is that if the current all-or-nothing action is somehow derailed before it reaches its call to COMMIT, the new version it has created is invisible to invokers of READ_CURRENT_VALUE. (They are also invisible to the all-or-nothing action that wrote them. Since it is sometimes convenient for an all-or-nothing action to read something that it has tentatively written, a separate procedure with a different test on line 8, named READ_MY_PENDING_VALUE, could do that.) Moreover if, for example, all-or-nothing action 99 crashes while partway through changing the values of nineteen different

```
procedure READ_CURRENT_VALUE (data_id, caller_id)
{
    starting at end of data_id repeat until beginning
    {
        v ← previous version of data_id;      // Get next older version
        a ← v.action_id;                      // Identify the action a that created it
        s ← a.outcome_record.state;           // Check action a's outcome record
        if s = COMMITTED then
            return v.value;
        else skip v;                          // Continue backward search
    }
    signal ("Tried to read an uninitialized variable!");
}

procedure WRITE_NEW_VALUE (reference data_id, new_value, caller_id)
{
    if caller_id.outcome_record.state = PENDING
    {
        append new version v to data_id;
        v.value ← new_value;
        v.action_id ← caller_id;
    }
    else signal ("Tried to write outside of an all-or-nothing action!");
}
```

Figure 9–15: Algorithms followed by READ_CURRENT_VALUE and WRITE_NEW_VALUE. The parameter *caller_id* is the action identifier returned by NEW_ACTION. In this version, only WRITE_NEW_VALUE uses *caller_id*. Later, READ_CURRENT_VALUE will also use it.

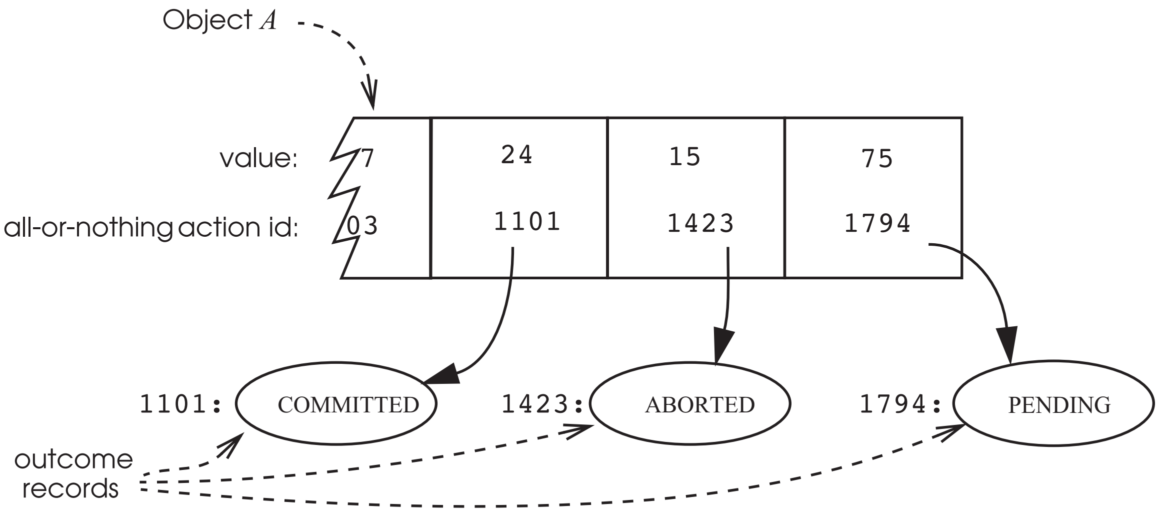


Figure 9–14: Portion of a version history, with outcome records. Some thread has recently called WRITE_NEW_VALUE specifying *data_id = A*, *new_value = 75*, and *client_id = 1794*. A caller to READ_CURRENT_VALUE will read the value 24 for A.

```
procedure TRANSFER (reference debit_account, reference credit_account, amount)
{
    my_id ← NEW_ACTION ();
    xvalue ← READ_CURRENT_VALUE (debit_account, my_id);
    xvalue ← xvalue - amount;
    WRITE_NEW_VALUE (debit_account, xvalue, my_id);
    yvalue ← READ_CURRENT_VALUE (credit_account, my_id);
    yvalue ← yvalue + amount;
    WRITE_NEW_VALUE (credit_account, yvalue, my_id);
    if xvalue > 0 then
        COMMIT (my_id);
    else
    {
        ABORT (my_id);
        signal ("Negative transfers are not allowed.");
    }
}
```

Figure 9–16: An all-or-nothing TRANSFER procedure, based on journal storage. (This program assumes that it is the only running thread. Isolating the transfer procedure from other threads that might be updating the same accounts concurrently requires additional mechanism that is discussed later in this chapter.)

data objects, all nineteen changes would be invisible to later invokers of READ_CURRENT_VALUE. If all-or-nothing action 99 does reach its call to COMMIT, that call commits the entire set of changes simultaneously and atomically, at the instant that it changes the outcome record from PENDING to COMMITTED. Pending versions would also be invisible to any concurrent action that reads data with READ_CURRENT_VALUE, a feature that will prove useful when we introduce concurrent threads and discuss isolation atomicity, but for the moment our only concern is that a system crash may prevent the current thread from committing or aborting, and we want to make sure that a later thread doesn't encounter partial results. As in the case of the calendar manager (see page 9–30), we assume that when a crash occurs, any all-or-nothing action that was in progress at the time was being supervised by some outside agent who realizes that a crash has occurred, uses READ_CURRENT_VALUE to find out what happened and if necessary initiates a replacement all-or-nothing action.

Figure 9–16 shows the TRANSFER procedure of page 9–21 reprogrammed as an all-or-nothing (but not, for the moment, isolated) action using the version history mechanism. This implementation of TRANSFER is more elaborate than the earlier one—it tests to see whether or not the account to be debited has enough funds to cover the transfer and if not it aborts the action. The order of steps in the transfer procedure is remarkably unconstrained by any consideration other than calculating the correct answer. The reading of *credit_account*, for example, could casually be moved to any point between NEW_ACTION and the place where *yvalue* is recalculated. We conclude that the journal storage system has made the pre-commit discipline much less onerous than we might have expected.

There is still one loose end: it is essential that updates to a version history and changes to an outcome record be all-or-nothing. That is, if the system fails while the thread is inside WRITE_NEW_VALUE, adjusting structures to append a new version, or inside COMMIT while updating the outcome record, the cell being written must not be muddled; it must either stay as it was before the crash or change to the intended new value. The solution is to design all modifications to the internal structures of journal storage so that they can be done by overwriting a single cell. For example, suppose that the name of a variable that has a version history refers to a cell that contains the address of the newest version, and that versions are linked from the newest version backwards, by address references. Adding a version consists of allocating space for a new version, reading the current address of the prior version, writing that address in the backward link field of the new version, and then updating the descriptor with the address of the new version. That last update can be done by overwriting a single cell. Similarly, updating an outcome record to change it from PENDING to COMMITTED can be done by overwriting a single cell.

As a first bootstrapping step, we have reduced the general problem of creating all-or-nothing actions to the specific problem of doing an all-or-nothing overwrite of one cell. As the remaining bootstrapping step, recall that we already know two ways to do a single-cell all-or-nothing overwrite: apply the ALL_OR_NOTHING_PUT procedure of figure 9–7. (If there is concurrency, updates to the internal structures of the version history also need to be isolated. Section D will explore methods of providing the necessary isolation atomicity.)

4. How version histories are used

The careful reader will note two possibly puzzling things about the version history scheme just described. Both will become less puzzling when we discuss concurrency and isolation in section D of this chapter:

1. Because READ_CURRENT_VALUE skips over any version belonging to another all-or-nothing action whose OUTCOME record is not COMMITTED, it isn't really necessary to change the OUTCOME record when an all-or-nothing action aborts; the record could just remain in the PENDING state indefinitely. However, when we introduce concurrency, we will find that a pending action may prevent other threads from reading variables for which the pending action created a new version, so it will become important to distinguish aborted actions from those that really are still pending.

2. As we have defined READ_CURRENT_VALUE, versions older than the most recent committed version are inaccessible and they might just as well be discarded. Discarding could be accomplished either as an additional step in the journal storage manager, or as part of a separate garbage collection activity. Alternatively, those older versions may be useful as an historical record, known as an *archive*, with the addition of timestamps on commit records and procedures that can locate and return old values created at specified times in the past. For this reason, a version history system is sometimes called a *temporal data base* or is said to provide *time domain addressing*. The banking industry abounds in requirements that make use of history information, such as reporting a consistent sum of balances in all bank accounts, paying interest on the fifteenth on balances as of the first of the month, or calculating the average balance last month. Another reason for not discarding old versions immediately will emerge when we discuss concurrency and isolation atomicity: concurrent threads may, for correctness, need to read old versions even after new versions have been created and committed.

Direct implementation of a version history raises concerns about performance: rather than simply reading a named storage cell, one must instead make at least one indirect reference through a descriptor that locates the storage cell containing the current version. If the cell storage device is on a magnetic disk, this extra reference is a potential bottleneck, though it can be alleviated with a cache. A bottleneck that is harder to alleviate occurs on updates. Whenever an application writes a new value, the journal storage layer must allocate space in unused cell storage, write the new version, and update the version history descriptor so that future readers can find the new version. Several disk writes are likely to be required. These extra disk writes may be hidden inside the journal storage layer and with added cleverness may be delayed until commit and batched, but they still have a cost. When storage access delays are the performance bottleneck, extra accesses slow things down.

In consequence, version histories are used primarily in low-performance applications. One common example is found in revision management systems used to coordinate teams doing program development. A programmer "checks out" a group of files, makes changes, and then "checks in" the result. The check-out and check-in operations are all-or-nothing and check-in makes each changed file the latest version in a complete history of that file, in case a problem is discovered later. (The check-in operation also verifies that no one else changed the files while they were checked out, which catches some, but not all, coordination errors.) A second example is that some interactive applications such as word processors or image editing systems provide a "deep undo" feature, which allows a user who decides that his or her recent editing is misguided to step backwards to reach an earlier, satisfactory state. A third example appears in file systems that automatically create a new version every time any application opens an existing file for writing; when the application closes the file, the file system tags a number suffix to the name of the previous version of the file and moves the original name to the new version. These interfaces employ version histories because users find them easy to understand and they provide all-or-nothing atomicity in the face of both system failures and user mistakes. Most such applications also provide an archive that is useful for reference and that allows going back to a known good version.

Applications requiring high performance are a different story. They, too, require all-or-nothing atomicity, but they usually achieve it by applying a specialized technique called a *log*. Logs are our next topic.

C. All-or-nothing atomicity II: Pragmatics

Database management applications such as airline reservation systems or banking systems usually require high performance as well as all-or-nothing atomicity, so their designers use streamlined atomicity techniques. The foremost of these techniques sharply separates the reading and writing of data from the failure recovery mechanism. The idea is to minimize the number of storage accesses required for the most common activities (application reads and updates). The trade-off is that the number of storage accesses for rarely-performed activities (failure recovery, which one hopes is actually exercised only occasionally, if at all) may not be minimal. The technique is called *logging*. Logging is also used for purposes other than atomicity, several of which sidebar 9–4 on page 9–48 describes.

1. Atomicity logs

The basic idea behind atomicity logging is to combine the all-or-nothing atomicity of journal storage with the speed of cell storage, by having the application twice record every change to data. The application first *logs* the change in journal storage, and then it *installs* the change in cell storage. One might think that writing data twice must be more expensive than writing it just once into a version history, but the separation permits specialized optimizations that can make the overall system faster.

The first recording, to journal storage, is optimized for fast writing by creating a single, interleaved version history of all variables, known as a *log*. The information describing each data update forms a record that the application appends to the end of the log. Since there is only one log, a single pointer to the end of the log is all that is needed to find the place to append the record of a change of any variable in the system. If the log medium is magnetic disk, and the disk is used only for logging, and the disk storage management system allocates sectors contiguously, the disk seek arm will need to move only when a disk cylinder is full, thus eliminating most seek delays. As we shall see, recovery does involve scanning the log, which is expensive, but recovery should be a rare event. Using a log is thus an example of following the hint to *optimize for the common case*.

The second recording, to cell storage, is optimized to make reading fast: the application installs by simply overwriting the previous cell storage record of that variable. The record kept in cell storage can be thought of as a cache that, for reading, bypasses the effort that would be otherwise be required to locate the latest version in the log. In addition, by not reading from the log the logging disk's seek arm can remain in position, ready for the next update. Figure 9–17 illustrates this two-step procedure.

The underlying idea is that the log is the authoritative record of the outcome of the action. Cell storage is merely a reference copy; if it is lost, it can be reconstructed from the log. The purpose of installing a copy in cell storage is to make both logging and reading faster.

Sidebar 9–4: The many uses of logs

A log is an object whose primary usage method is to append a new record. Log implementations normally provide procedures to read entries from oldest to newest or in reverse order, but there is usually not any procedure for modifying previous entries. Logs are used for several quite distinct purposes, and this range of purposes sometimes gets confused in real-world designs and implementations. Here are some of the most common uses for logs:

1. *Atomicity log*. If one logs the component actions of an all-or-nothing action, together with sufficient before and after information, then a crash recovery procedure can undo (and thus roll back the effects of) all-or-nothing actions that didn't get a chance to complete, or finish all-or-nothing actions that committed but that didn't get a chance to record all of their effects.

2. *Archive log*. If the log is kept indefinitely, it becomes a place where old values of data and the sequence of actions taken by the system or its applications can be kept for review. There are many uses for archive information: watching for failure patterns, reviewing the actions of the system preceding and during a security breach, recovery from application-layer mistakes (e.g., a clerk incorrectly deleted an account), historical study, fraud control, and compliance with record-keeping requirements.

3. *Performance log*. Most mechanical storage media have much higher performance for sequential access than for random access. Since logs are written sequentially, they are ideally suited to such storage media. It is possible to take advantage of this match to the physical properties of the media by structuring data to be written in the form of a log. When combined with a cache that eliminates most disk reads, a performance log can provide a significant speed-up. As will be seen in the accompanying text, an atomicity log is usually also a performance log.

4. *Durability log*. If the log is stored on a non-volatile medium—say magnetic tape—that fails in ways and at times that are independent from the failures of the cell storage medium—which might be magnetic disk—then the copies of data in the log are replicas that can be used as backup in case of damage to the copies of the data in cell storage. This kind of log helps implement durable storage. Any log that uses a non-volatile medium, whether intended for atomicity, archiving or performance, typically also helps support durability.

It is essential to have these various purposes—all-or-nothing atomicity, archive, performance, and durable storage—distinct in one's mind when examining or designing a log implementation, because they lead to different priorities among design trade-offs. When archive is the goal, low cost of the storage medium is usually more important than quick access, because archive logs are large but, in practice, infrequently read. When durable storage is the goal, it may be important to use storage media with different physical properties, so that failure modes will be as independent as possible. When all-or-nothing atomicity or performance is the purpose, minimizing mechanical movement of the storage device becomes a high priority. Because of the competing objectives of different kinds of logs, as a general rule, it is usually a wise move to implement separate, dedicated logs for different functions.

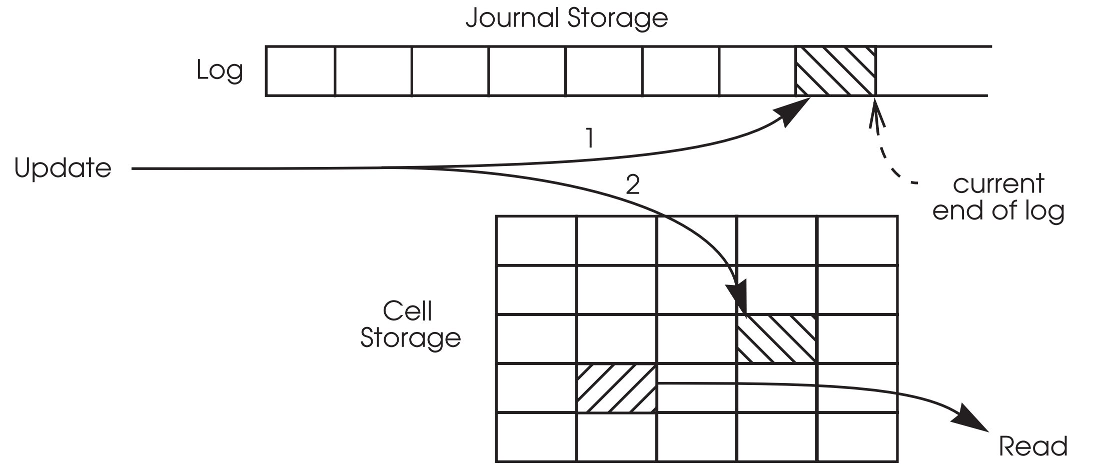


Figure 9–17: Logging for all-or-nothing atomicity. The application performs updates by first appending a record of the update to the log in journal storage, and then installing the update in cell storage by overwriting. The application reads only from cell storage.

By recording data twice, we obtain high performance in writing, high performance in reading, and all-or-nothing atomicity, all at the same time.

There are three common logging configurations, shown in figure 9–18. In each of these three configurations, the log resides in non-volatile storage. For the *in-memory database*, cell storage resides entirely in some volatile storage medium. In the second common configuration, cell storage resides in non-volatile storage along with the log. Finally, high-performance database management systems usually blend the two preceding configurations by implementing a cache for cell storage in a volatile medium, and a potentially independent multilevel memory management algorithm moves data between the cache and non-volatile cell storage.

Recording everything twice adds one significant complication to all-or-nothing atomicity, because the system can crash between the time a change is logged and the time it is installed. To maintain all-or-nothing atomicity, logging systems follow a protocol that has two fundamental requirements. The first requirement is a constraint on the order of logging and installing. The second requirement is to run an explicit *recovery* procedure after every crash. (We saw a preview of the strategy of using a recovery procedure in figure 9–7, which used a recovery procedure named CHECK_AND_REPAIR.)

2. *Logging protocols*

There are several kinds of atomicity logs that vary in the order in which things are done and in the details of information logged. However, all of them involve the ordering constraint implied by the numbering of the arrows in figure 9–17. The constraint is a version of the

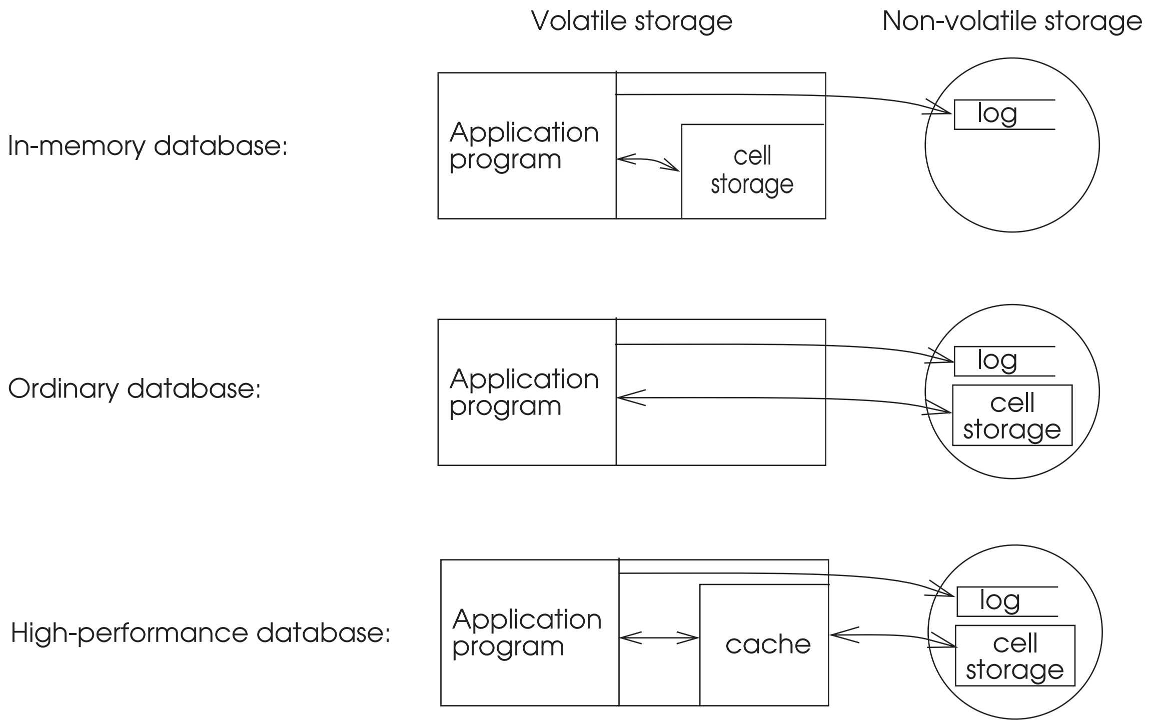


Figure 9–18: Three common logging configurations. Arrows show data flow as the application reads, logs, and installs data.

golden rule of atomicity (never modify the only copy), known as the *write-ahead-log* (WAL) protocol:

Write-ahead-log protocol

Log the update *before* installing it.

The reason is that logging appends but installing overwrites. If an application violates this protocol by installing an update before logging it and then for some reason must abort, or the system crashes, there is no systematic way to discover the installed update and, if necessary, reverse it. The write-ahead-log protocol ensures that if a crash occurs, a recovery procedure can, by consulting the log, systematically find all completed and intended changes to cell storage and either restore those records to old values or set them to new values, as appropriate to the circumstance.

The basic element of an atomicity log is the *log record*. Before an action that is to be all-or-nothing installs a data value, it appends to the end of the log a new record of type CHANGE containing, in the general case, three pieces of information (we shall later see special cases that allow omitting item 2 or item 3):

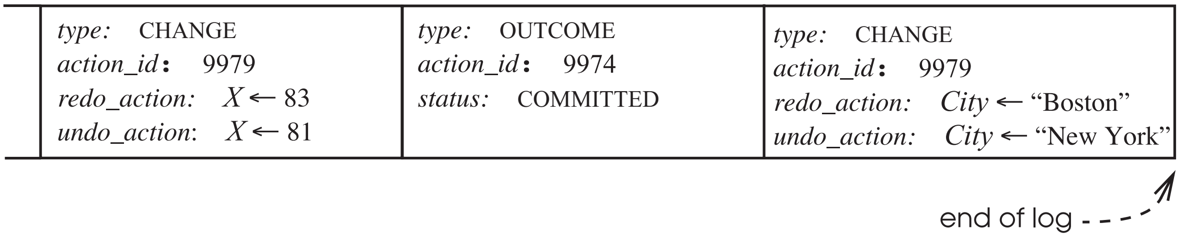
1. The identity of the all-or-nothing action that is performing the update.

2. A component action that, if performed, installs the intended value in cell storage. This component action is a kind of an insurance policy in case the system crashes. If the all-or-nothing action commits, but then the system crashes before the action has a chance to perform the install, the recovery procedure can perform the install on behalf of the action. Some systems call this component action the *do* action, others the *redo* action. For mnemonic compatibility with item 3, this text calls it the redo action.

3. A second component action that, if performed, reverses the effect on cell storage of the planned install. This component action is known as the *undo* action because if, after doing the install, the all-or-nothing action aborts or the system crashes, it may be necessary for the recovery procedure to reverse the effect of (*undo*) the install.

An application appends a log record by invoking the lower-layer procedure LOG, which itself must be atomic. The LOG procedure is another example of bootstrapping: Starting with, for example, the ALL_OR_NOTHING_PUT described earlier in this chapter, a log designer creates a generic LOG procedure, and using the LOG procedure an application programmer then can implement all-or-nothing atomicity for any properly designed composite action.

Upon undertaking an all-or-nothing action, the first step is to log a BEGIN record that contains just the new all-or-nothing action's identity. As the all-or-nothing action proceeds through its pre-commit phase, it logs CHANGE records. Upon deciding to commit or abort, the all-or-nothing action logs an OUTCOME record that becomes the authoritative indication of the outcome of the all-or-nothing action. The instant that the all-or-nothing action logs the OUTCOME record is its commit point. Because the log is the authoritative record of the action, the all-or-nothing action can perform installs to cell storage at any convenient time that is consistent with the write-ahead-log protocol, either before or after logging the OUTCOME record. The final step of an action is to log an END record, again containing just the action's identity, to show that the action has completed all of its installs. (Logging all four kinds of activity—BEGIN, CHANGE, OUTCOME, and END—is more general than sometimes necessary. As we shall see, some logging systems can combine, e.g., OUTCOME and END, or BEGIN with the first CHANGE.) Figure 9–19 shows examples of the last three records of a log, two typical CHANGE records of one all-or-nothing action, separated by the OUTCOME record of a different all-or-nothing action.



type: CHANGE
action_id: 9979
redo_action: $X \leftarrow 83$
undo_action: $X \leftarrow 81$

type: OUTCOME
action_id: 9974
status: COMMITTED

type: CHANGE
action_id: 9979
redo_action: *City* ← "Boston"
undo_action: *City* ← "New York"

Figure 9–19: An example of the end of an atomicity log, containing two CHANGE records for one all-or-nothing action and the OUTCOME record of a different all-or-nothing action.

One consequence of installing results in cell storage is that for an all-or-nothing action to abort it may have to do some clean-up work. Moreover, if the system involuntarily terminates a thread that is in the middle of an all-or-nothing action (because, for example, the thread has gotten into a deadlock or an endless loop) some entity other than the hapless thread must clean things up. If this clean-up step were omitted, the all-or-nothing action could remain pending indefinitely. The system cannot simply ignore indefinitely pending actions, because all-or-nothing actions initiated by other threads are likely to want to use the data that the terminated action changed. (This is actually an isolation concern, one of the places where all-or-nothing atomicity and isolation atomicity intersect.)

If the action being aborted did any installs, those installs are still in cell storage, so simply appending to the log an OUTCOME record saying that the action aborted is not enough to make it appear to later observers that the all-or-nothing action did nothing. The solution to this problem is to execute a generic ABORT procedure. The ABORT procedure restores to their old values all cell storage variables that the all-or-nothing action installed. The ABORT procedure simply scans the log backwards looking for log entries created by this all-or-nothing action; for each CHANGE record it finds, it performs the logged *undo_action,* thus restoring the old values in cell storage. The backward search terminates when the ABORT procedure finds that all-or-nothing action's BEGIN record. Figure 9–20 illustrates.

```
procedure ABORT (action_id)
{
    starting at end of log repeat until beginning
    {
        log_record ← previous record of log
        if log_record.id = action_id then
        {
            if (log_record.type = OUTCOME)
                then signal ("Can't abort an already completed action.");
            if (log_record.type = CHANGE)
                then perform undo_action of log_record;
            if (log_record.type = BEGIN)
                then break repeat;
        }
    }
    LOG (action_id, OUTCOME, ABORTED);          // Block future undos.
    LOG (action_id, END);
    return;
}
```

Figure 9–20: Generic ABORT procedure for a logging system. The argument *action_id* identifies the action to be aborted. An atomic action calls this procedure if it decides to abort. In addition, the operating system may call this procedure if it decides to terminate the action, for example to break a deadlock or because the action is running too long. The LOG procedure must itself be atomic.

The extra work required to undo cell storage installs when an all-or-nothing action aborts is another example of *optimizing for the common case*: one expects that most all-or-nothing actions will commit, and that aborted actions should be relatively rare. The extra effort of an occasional roll back of cell storage values will (one hopes) be more than repaid by the more frequent gains in performance on updates, reads, and commits.

3. Recovery procedures

The write-ahead log protocol is the first of the two required protocol elements of a logging system. The second required protocol element is that, following every system crash, the system must run a recovery procedure before it allows ordinary applications to use the data. The details of the recovery procedure depend on the particular configuration of the journal and cell storage with respect to volatile and non-volatile memory.

Consider first recovery for the in-memory database of figure 9–18. Since a system crash may corrupt anything that is in volatile memory, including both the state of cell storage and the state of any currently running threads, restarting a crashed system usually begins by resetting all volatile memory. The effect of this reset is to abandon both the cell storage version of the database and any all-or-nothing actions that were in progress at the time of the crash. On the other hand, the log, since it resides on non-volatile journal storage, is unaffected by the crash and should still be intact.

The simplest recovery procedure performs two passes through the log. On the first pass, it scans the log *backward* from the last record, so the first evidence it will encounter of each all-or-nothing action is the last record that the all-or-nothing action logged. A backward log scan is sometimes called a LIFO (for last-in, first-out) log review. As the recovery procedure scans backward, it collects in a set the identity and completion status of every all-or-nothing action that logged an OUTCOME record before the crash. These actions, whether committed or aborted, are known as *winners*.

When the backward scan is complete the set of winners is also complete, and the recovery procedure begins a forward scan of the log. The reason the forward scan is needed is that restarting after the crash completely reset the cell storage. During the forward scan the recovery procedure performs, in the order found in the log, all of the REDO actions of every winner whose OUTCOME record says that it COMMITTED. Those REDOs reinstall all committed values in cell storage, so at the end of this scan, the recovery procedure has restored cell storage to a desirable state. This state is as if every all-or-nothing action that committed before the crash had run to completion, while every all-or-nothing action that aborted or that was still pending at crash time had never existed. The database system can now open for regular business. Figure 9–21 illustrates.

This recovery procedure emphasizes the point that a log can be viewed as an authoritative version of the entire database, sufficient to completely reconstruct the reference copy in cell storage.

There exist cases for which this recovery procedure may be overkill, when the durability requirement of the data is minimal. For example, the all-or-nothing action may have been to make a group of changes to soft state in volatile storage. If the soft state is completely lost in a crash, there would be no need to redo installs, because the definition of soft state is that the application is prepared to construct new soft state following a crash. Put another way, given the options of “all” or “nothing,” when the data is all soft state “nothing” is always an appropriate outcome after a crash.

```
procedure RECOVER ()
{                                   // Recovery procedure for a volatile, in-memory database.
    winners ← NULL;
    starting at end of log repeat until beginning
    {
        log_record ← previous record of log
        if (log_record.type = OUTCOME)
            then winners ← winners + log_record;      // Set addition.
    }
    starting at beginning of log repeat until end
    {
        log_record ← next record of log
        if (log_record.type= CHANGE)
            and (log_record.action_id is in winners)
            and (log_record.action_id.status = COMMITTED) then
            perform log_record.redo_action;
    }
    return;
}
```

Figure 9–21: An idempotent redo-only recovery procedure for an in-memory database. Because RECOVER writes only to volatile storage, if a crash occurs while it is running it is safe to run it again.

A critical design property of the recovery procedure is that, if there should be another system crash during recovery, it must still be possible to recover. Moreover, it must be possible for any number of crash-restart cycles to occur without compromising the correctness of the ultimate result. The method is to design the recovery procedure to be *idempotent*. That is, design it so that if it is interrupted and restarted from the beginning it will produce exactly the same result as if it had run to completion to begin with. With the in-memory database configuration, this goal is an easy one: just make sure that the recovery procedure modifies only volatile storage. Then, if a crash occurs during recovery, the loss of volatile storage automatically restores the state of the system to the way it was when the recovery started, and it is safe to run it again from the beginning. If the recovery procedure ever finishes, the state of the cell storage copy of the database will be correct, no matter how many interruptions and restarts intervened.

The ABORT procedure similarly needs to be idempotent, because if an all-or-nothing action decides to abort and, while running ABORT, some timer expires, the system may decide to terminate and call ABORT for that same all-or-nothing action. The version of abort in figure 9–20 will satisfy this requirement if the individual undo actions are themselves idempotent.

4. *Other logging configurations: non-volatile cell storage*

Placing cell storage in volatile memory is a <u>sweeping simplification</u> that works well for small and medium-sized databases, but some databases are too large for that to be practical, so the designer finds it necessary to plaace cell storage on some cheaper, non-volatile storage medium such as magnetic disk, as in the second configuration of figure 9–18. But with a non-volatile storage medium, installs survive system crashes, so the simple recovery procedure used with the in-memory database would have two shortcomings:

1. If, at the time of the crash, there were some pending all-or-nothing actions that had installed changes, those changes will survive the system crash. The recovery procedure must reverse the effects of those changes, just as if those actions had aborted.

2. That recovery procedure reinstalls the entire database, even though in this case much of it is probably intact in non-volatile storage. If the database is large enough that it requires non-volatile storage to contain it, the cost of unnecessarily reinstalling it in its entirety at every recovery is likely to be unacceptable.

In addition, reads and writes to non-volatile cell storage are likely to be slow, so it is nearly always the case that the designer installs a cache in volatile memory, along with a multilevel memory manager, thus moving to the third configuration of figure 9–18. But that addition introduces yet another shortcoming:

3. In a multilevel memory system, the order in which data is written from volatile levels to non-volatile levels is generally under control of a multilevel memory manager, which may, for example, be running a least-recently-used algorithm. As a result, at the instant of the crash some things that were thought to have been installed may not yet have migrated to the non-volatile memory.

To postpone consideration of this shortcoming, let us temporarily assume that the multilevel memory manager implements a write-through cache. (Subsection C.6, below, will return to the case where the cache is not write-through.) With a write-through cache, we can be certain that everything that the application program has installed has been written to non-volatile storage. This assumption drops the third shortcoming out of our list of concerns and the situation is the same as if we were using the “Ordinary Database” configuration of figure 9–18 with no cache. But we still have to do something about the first two shortcomings, and we also must make sure that the modified recovery procedure is still idempotent.

To address the first shortcoming, that the database may contain installs from actions that should be undone, we need to modify the recovery procedure of figure 9–21. As the recovery procedure performs its initial backward scan, rather than looking for winners, it instead collects in a set the identity of those all-or-nothing actions that were still in progress at the time of the crash. The actions in this set are known as *losers*, and they can include both actions that committed and actions that did not. Losers are easy to identify because the first log record that contains their identity that is encountered in a backward scan will be something other than an END record. To identify the losers, the pseudocode keeps track of which actions logged an END record in an auxiliary list named *completeds*. When recover comes across a log record belong to an action that is not in *completed*, it adds that action to the set named *losers*. In addition, as it scans backwards, whenever the recovery procedure encounters a CHANGE record belonging to a loser, it performs the UNDO action listed in the record. In the course of the LIFO log review, all of the installs performed by losers will thus be rolled back and the state of the cell storage will be as if the all-or-nothing actions of losers had never started. Next, RECOVER performs the forward log scan of the log, performing the redo actions of the all-or-nothing actions that committed, as shown in figure 9–22. Finally, the recovery

```
procedure RECOVER ()
{                                          // Recovery procedure for non-volatile cell memory
    completeds ← NULL;
    losers ← NULL;
    starting at end of log repeat until beginning
    {
        log_record ← previous record of log
        if (log_record.type = END)
            then completeds ← completeds + log_record;        // Set addition.
        if (log_record.action_id is not in completeds) then
        {
            losers ← losers + log_record;                  // Add if not already in set.
            if (log_record.type = CHANGE) then
                perform log_record.undo_action;
        }
    }
    starting at beginning of log repeat until end
    {
        log_record ← next record of log
        if (log_record.type = CHANGE)
            and (log_record.action_id.status = COMMITTED) then
            perform log_record.redo_action;
    }
    for each log_record in losers do
        log (log_record.action_id, END);                   // Show action completed.
    return;
}
```

Figure 9–22: An idempotent undo/redo recovery procedure for a system that performs installs to non-volatile cell memory. In this recovery procedure, *losers* are all-or-nothing actions that were in progress at the time of the crash.

procedure logs an END record for every all-or-nothing action in the list of losers. This END record transforms the loser into a completed action, thus ensuring that future recoveries will ignore it and not perform its undos again. For future recoveries to ignore aborted losers is not just a performance enhancement, it is essential, to avoid incorrectly undoing updates to those same variables made by future all-or-nothing actions.

As before, the recovery procedure must be idempotent, so that if a crash occurs during recovery the system can just run the recovery procedure again. In addition to the technique used earlier of placing the temporary variables of the recovery procedure in volatile storage, each individual undo action must also be idempotent. For this reason, both redo and undo actions are usually expressed as *blind writes*. A blind write is a simple overwriting of a data value without reference to its previous value. Because blind writes are inherently idempotent, no matter how many times one repeats it, the result is always the same. Thus, if a crash occurs part way through the logging of END records of losers, immediately rerunning the recovery procedure will still leave the database correct. Any losers that now have END records will be treated as completed on the rerun, but that is OK because the previous attempt of the recovery procedure has already undone their installs.

```
procedure RECOVER ()
{                                                               // Recovery procedure for rollback recovery.
    completeds ← NULL;
    losers ← NULL;
    starting at end of log repeat until beginning                        // Perform undo scan.
    {
        log_record ← previous record of log
        if (log_record.type = OUTCOME)
            then completeds ← completeds + log_record;                   // Set addition.
        if (log_record.action_id is not in completeds) then
        {
            losers ← losers + log_record;                                // Must be a new loser.
            if (log_record.type = CHANGE) then
                perform log_record.undo_action;
        }
    }
    for each log_record in losers do
        log (log_record.action_id, OUTCOME, ABORT);                      // Block future undos.
    return;
}
```

Figure 9–23: An idempotent undo-only recovery procedure for a rollback logging system.

As for the second shortcoming, that the recovery procedure unnecessarily redoes every install, even installs not belong to losers, we can significantly simplify (and speed up) recovery by analyzing why we have to redo any installs at all. The reason is that, although the WAL protocol requires logging of changes to occur before install, there is no necessary ordering between commit and install. Until a committed action logs its END record, there is no assurance that any particular install of that action has actually happened yet. On the other hand, any committed action that has logged an END record has completed its installs. The conclusion is that the recovery procedure does not need to redo installs for any committed action that has logged its END record. A useful exercise is to modify the procedure of figure 9–22 to take advantage of that observation.

It would be even better if the recovery procedure never had to redo *any* installs. We can arrange for that by placing another requirement on the application: it must perform all of its installs *before* it logs its OUTCOME record. That requirement, together with the write-through cache, ensures that the installs of every completed all-or-nothing action are safely in non-volatile cell storage and there is thus never a need to perform *any* redo actions. (It also means that there is no need to log an END record.) The result is that the recovery procedure needs only to undo the installs of losers, and it can skip the entire forward scan, leading to the simpler recovery procedure of figure 9–23. This scheme, because it requires only undos, is sometimes called *undo logging* or *rollback recovery.* A property of rollback recovery is that for completed actions, cell storage is just as authoritative as the log. As a result, one can garbage collect the log, discarding the log records of completed actions. The now much smaller log may then be able to fit in a faster storage medium for which the durability requirement is only that it outlast pending actions.

There is an alternative, symmetric constraint used by some logging systems. Rather than requiring that all installs be done *before* logging the OUTCOME record, one can instead require that all installs be done *after* recording the OUTCOME record. With this constraint, the set of CHANGE records in the log that belong to that all-or-nothing action become a description of its intentions. If there is a crash before logging an OUTCOME record, we are assured that no installs have happened, so the recovery never needs to perform any undos. On the other hand, it may have to perform installs for all-or-nothing actions that committed. This scheme is called *redo logging* or *roll-forward recovery*. Furthermore, because we are uncertain about which installs actually have taken place, the recovery procedure must perform *all* logged installs for all-or-nothing actions that did not log an END record. Any all-or-nothing action that logged an END record must have completed all of its installs, so there is no need for the recovery procedure to perform them. The recovery procedure thus reduces to doing installs just for all-or-nothing actions that were interrupted between the logging of their OUTCOME and END records. Recovery with redo logging can thus be quite swift, though it does require both a backward and forward scan of the entire log.

We can summarize the procedures for atomicity logging as follows:

- Log to journal storage before installing in cell storage (WAL protocol)

- If all-or-nothing actions perform *all* installs to non-volatile storage before logging their OUTCOME record, then recovery needs only to undo the installs of incomplete uncommitted actions. (rollback/undo recovery)

- If all-or-nothing actions perform *no* installs to non-volatile storage before logging their OUTCOME record, then recovery needs only to redo the installs of incomplete committed actions. (roll-forward/redo recovery)

- If all-or-nothing actions are not disciplined about when they do installs to non-volatile storage, then recovery needs to both redo the installs of incomplete committed actions *and* undo the installs of incomplete uncommitted ones.

In addition to reading and updating memory, an all-or-nothing action may also need to send messages, for example, to report its success to the outside world. The action of sending a message is just like any other component action of the all-or-nothing action. To provide all-or-nothing atomicity, message sending can be handled in a way analogous to memory update. That is, log a CHANGE record with a redo action that sends the message. If a crash occurs after the all-or-nothing action commits, the recovery procedure will perform this redo action along with other redo actions that perform installs. In principle, one could also log an *undo_action* that sends a compensating message ("Please ignore my previous communication!"). However, an all-or-nothing action will usually be careful not to actually send any messages until after the action commits, so roll-forward recovery applies. For this reason, a designer would not normally specify an undo action for a message or for any other action that has outside-world visibility such as printing a receipt, opening a cash drawer, drilling a hole, or firing a missile.

Incidentally, although much of the professional literature about data base atomicity and recovery uses the terms "winner" and "loser" to describe the recovery procedure, different recovery systems use subtly different definitions for the two sets, depending on the exact logging scheme, so it is a good idea to review those definitions carefully.

5. Checkpoints

Constraining the order of installs to be all before or all after the logging of the OUTCOME record is not the only thing we could do to speed up recovery. Another technique that can shorten the log scan is to occasionally write some additional information, known as a *checkpoint,* to non-volatile storage. Although the principle is always the same, the exact information that is placed in a checkpoint varies from one system to another. A checkpoint can include information written either to cell storage or to the log (where it is known as a *checkpoint record*) or both.

Suppose, for example, that the logging system maintains in volatile memory a list of identifiers of all-or-nothing actions that have started but have not yet recorded an END record, together with their pending/committed/aborted status, keeping it up to date by observing logging calls. The logging system then occasionally logs this list as a CHECKPOINT record. When a crash occurs sometime later, the recovery procedure begins a LIFO log scan as usual, collecting the sets of completed actions and losers. When it comes to a CHECKPOINT record it can immediately fill out the set of losers by adding those all-or-nothing actions that were listed in the checkpoint that did not later log an END record. This list may include some all-or-nothing actions listed in the CHECKPOINT record as COMMITTED, but that did not log an END record by the time of the crash. Their installs still need to be performed, so they need to be added to the set of losers. The LIFO scan continues, but only until it has found the BEGIN record of every loser.

With the addition of CHECKPOINT records, the recovery procedure becomes more complex, but is potentially shorter in time and effort:

1. Do a LIFO scan of the log back to the last CHECKPOINT record, collecting identifiers of losers and undoing all actions they logged.

2. Complete the list of losers from information in the checkpoint.

3. Continue the LIFO scan, undoing the actions of losers, until every BEGIN record belonging to every loser has been found.

4. Perform a forward scan from that point to the end of the log, performing any committed actions belonging to all-or-nothing actions in the list of losers that logged an OUTCOME record with status COMMITTED.

In systems in which long-running all-or-nothing actions are uncommon, step 3 will typically be quite brief or even empty, greatly shortening recovery. A good exercise is to modify the recovery program of figure 9–22 to accommodate checkpoints.

Checkpoints are also used with in-memory databases, to provide durability without the need to reprocess the entire log after every system crash. A useful checkpoint procedure for an in-memory database is to make a snapshot of the complete database, writing it to one of two alternating (for all-or-nothing atomicity) dedicated non-volatile storage regions, and then logging a CHECKPOINT record that contains the address of the latest snapshot. Recovery then involves scanning the log back to the most recent CHECKPOINT record, collecting a list of committed all-or-nothing actions, restoring the snapshot described there, and then performing redo actions of those committed actions from the CHECKPOINT record to the end of the log. The main challenge in this scenario is isolating the writing of the snapshot from any concurrent update activity. That can be done either by preventing all updates for the duration of the snapshot or by applying more complex isolation techniques such as those described in later sections of this chapter.

6. What if the cache is not write-through? (advanced topic)

Between the log and the write-through cache, the logging configurations just described require, for every data update, two synchronous writes to non-volatile storage, with attendant delays waiting for the writes to complete. Since the original reason for introducing a log was to increase performance, these two synchronous write delays usually become the system performance bottleneck. Designers who are interested in maximizing performance would prefer to use a cache that is not write-through, so that writes can be deferred until a convenient time when they can be done in batches. Unfortunately, the application then loses control of the order in which things are actually written to non-volatile storage. Loss of control of order has a significant impact on our all-or-nothing atomicity algorithms, since they require, for correctness, constraints on the order of writes and certainty about which writes have been done.

The first concern is for the log itself, because the write-ahead log protocol requires that appending a CHANGE record to the log precede the corresponding install in cell storage. One simple way to enforce the WAL protocol is to make just log writes write-through, but allow cell storage writes to occur whenever the cache manager finds it convenient. However, this relaxation means that if the system crashes there is no assurance that any particular install has actually migrated to non-volatile storage. The recovery procedure, assuming the worst, cannot take advantage of checkpoints and must again perform installs starting from the beginning of the log. To avoid that possibility, the usual design response is to flush the cache as part of logging each checkpoint record. Unfortunately, flushing the cache and logging the checkpoint must be done as a single atomic action that is isolated from concurrent updates, which creates another design challenge. This challenge is surmountable, but the complexity is increasing.

Some systems pursue performance even farther. A popular technique is to write the log to a volatile buffer, and *force* that entire buffer to non-volatile storage only when an all-or-nothing action commits. This strategy allows batching several CHANGE records with the next OUTCOME record in a single synchronous write. Although this step would appear to violate the write-ahead log protocol, that protocol can be restored by making the cache used for cell storage a bit more elaborate; its management algorithm must avoid writing back any install for which the corresponding log record is still in the volatile buffer. The trick is to *number* each log record in sequence, and tag each record in the cell storage cache with the sequence number of its log record. Whenever the system forces the log, it tells the cache manager the sequence number of the last log record that it wrote, and the cache manager is careful never to write back any cache record that is tagged with a higher log sequence number.

We have in this section seen some good examples of the *law of diminishing returns* at work: schemes that improve performance sometimes require significantly increased complexity. Before undertaking any such scheme, it is essential to evaluate carefully how much extra performance one stands to gain.

D. Isolation atomicity I: Concepts

The mechanisms developed in the previous sections of this chapter provide atomicity in the face of failure, so that other atomic actions that take place after the failure and subsequent recovery find that an interrupted atomic action apparently either executed all of its steps or none of them. This and the next section investigate how to also provide atomicity of concurrent actions, known as *isolation atomicity*. In this development we shall provide *both* all-or-nothing atomicity *and* isolation atomicity, so we will now be able to call the resulting atomic actions *transactions*.

Concurrency atomicity requires additional mechanism because when an atomic action installs data in cell storage, that data is immediately visible to all concurrent actions. Even though the version history mechanism can hide pending changes from concurrent atomic actions, they can read other variables that the first atomic action plans to change. Thus, the composite nature of a multiple-step atomic action may still be discovered by a concurrent atomic action that happens to look at the value of a variable in the midst of execution of the first atomic action. Thus, making an action atomic with respect to concurrent threads—that is, *isolating* it—requires further effort.

Recall that section A.4 defined isolation of concurrent actions to be correct *if every result is guaranteed to be one that could have been obtained by some purely serial application* of those same actions. So we are looking for techniques that guarantee to produce the same result as if concurrent actions had been applied serially.

In this section D we explore three successively better isolation schemes, using version histories, where "better" means that the scheme allows more concurrency. Each of these schemes has a straightforward and compelling correctness argument. In the following section E we examine a somewhat different approach, using locks, that can provide higher performance but for which correctness arguments are more difficult.

1. Achieving isolation: simple serialization

A version history assigns a unique identifier to each atomic action so that it can link tentative versions of variables to the action's outcome record. Suppose that we require that the unique identifiers be consecutive integers, which we interpret as serial numbers, and we modify BEGIN_TRANSACTION to create the procedure BEGIN_TRANSACTION by adding enforcement of the following *simple serialization* rule: each newly created transaction n must, before reading or writing any data, wait until the preceding transaction $n - 1$ has either committed or aborted. (To ensure that there is always a transaction $n - 1$, assume that the system was initialized by creating a transaction number zero with an OUTCOME record in the committed state.) Figure 9–24 shows this version of BEGIN_TRANSACTION. The scheme forces all transactions to execute in the serial order that threads happen to invoke BEGIN_TRANSACTION. Since that order is a possible serial order of the various transactions, by definition simple

```
procedure BEGIN_TRANSACTION ()
{
    id ← NEW_OUTCOME_RECORD (PENDING);   // Create, initialize, and assign id.
    previous_id ← id – 1;
    wait until previous_id.outcome_record.state ≠ PENDING;
    return id;
}
```

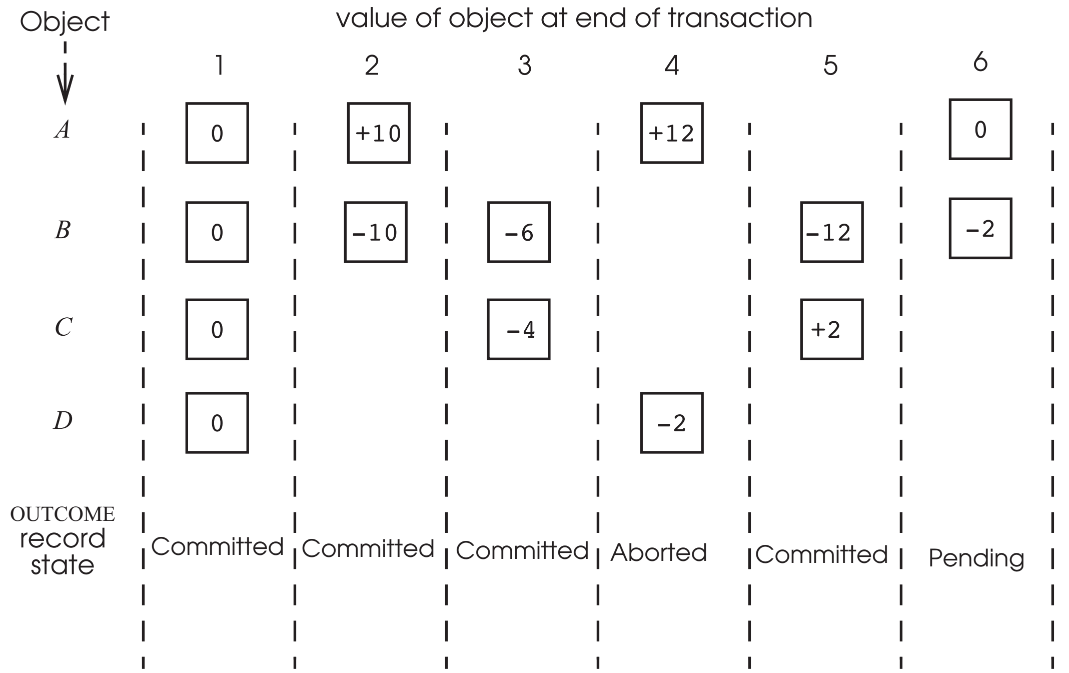
Figure 9–24: BEGIN_TRANSACTION with the simple serialization discipline to achieve isolation. In order that there be an *id* – 1 for every value of *id*, startup of the system must include creating a dummy transaction with *id* = 0 and *id.outcome_record.state* = COMMITTED. Pseudocode for the procedure NEW_OUTCOME_RECORD appears in figure 9–29.

serialization will produce transactions that are serialized and thus correctly isolated. Simple serialization trivially provides isolation, and the transaction is still all-or-nothing, so the transaction is now atomic both in the case of failure and in the presence of concurrency.

Simple serialization provides isolation by being too conservative: it forbids all concurrency among transactions, even if they would not interfere with one another. Nevertheless, this approach actually has some practical value—in some applications it may be just the right thing to do, on the basis of simplicity. Concurrent threads can do much of their work in parallel, because simple serialization comes into play only during those times that threads are executing transactions, which they generally would be only at the moments they are working with shared variables. If such moments are infrequent or if the transactions that need to be isolated all modify the same small set of shared variables, simple serialization is likely to be just about as effective as any other scheme. In addition, by looking carefully at why it works, we can discover less conservative approaches that allow more concurrency, yet still have compelling arguments that they preserve correctness. Put another way, the remainder of study of isolation techniques is fundamentally nothing but invention and analysis of increasingly effective—and increasingly complex—performance improvement measures.

The version history provides a useful representation for this analysis. Figure 9–25 illustrates in a single figure the version histories of a banking system consisting of four accounts named *A*, *B*, *C*, and *D*, during the execution of six transactions, with serial numbers 1 through 6. The first transaction initializes all the objects to contain the value 0 and the following transactions transfer various amounts back and forth between pairs of accounts.

This figure provides a straightforward interpretation of why simple serialization works correctly. Consider transaction 3, which must read and write objects *B* and *C* in order to transfer funds from one to the other. The way for transaction 3 to produce results as if it ran after transaction 2 is for all of 3's input objects to have values that include all the effects of transaction 2—if transaction 2 commits, then any objects it changed and that 3 uses should have new values; if transaction 2 aborts, then any objects it tentatively changed and 3 uses should contain the values that they had when transaction 2 started. Since in this example transaction 3 reads *B* and transaction 2 creates a new version of *B*, it is clear that for transaction 3 to produce a correct result it must wait until transaction 2 either commits or aborts. Simple serialization requires that wait, and thus assures correctness.



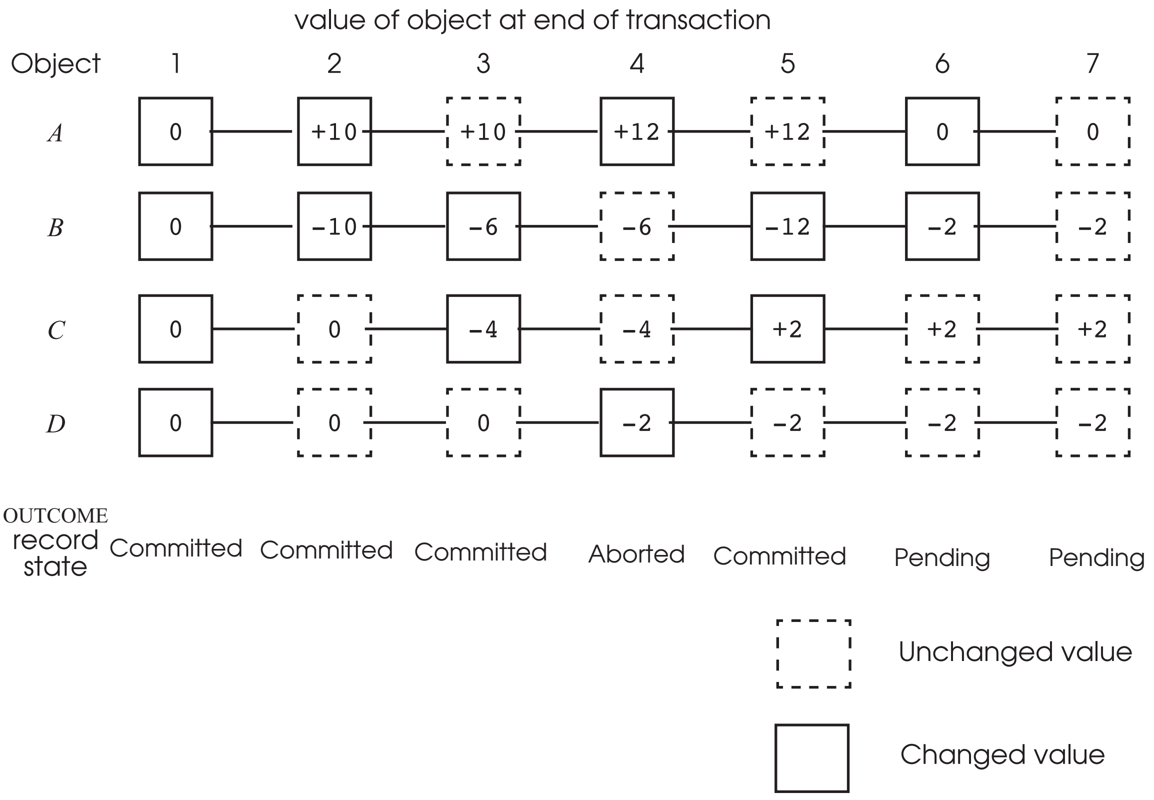
transaction 1: initialize all accounts to 0
2: transfer 10 from *B* to *A*
3: transfer 4 from *C* to *B*
4: transfer 2 from *D* to *A* (aborts)
5: transfer 6 from *B* to *C*
6: transfer 10 from *A* to *B*

Figure 9–25: Version history of a banking system.

Figure 9–25 also provides some clues about how to increase concurrency. Looking at transaction 4 (the example shows that transaction 4 will ultimately abort for some reason, but suppose we are just starting transaction 4 and don't know that yet), it is apparent that simple serialization is too strict. Transaction 4 reads values only from *A* and *D*, yet transaction 3 has no interest in either object. Thus the values of *A* and *D* will be the same whether or not transaction 3 commits, and a discipline that forces 4 to wait for 3's completion delays 4 unnecessarily. On the other hand, transaction 4 does use an object that transaction 2 modifies, so transaction 4 must wait for transaction 2 to complete. Of course, simple serialization guarantees that, since transaction 4 can't begin till transaction 3 completes and transaction 3 couldn't have started until transaction 2 completed.

These observations suggest that there may be other, more relaxed, disciplines that can still guarantee correct results. They also suggest that any such discipline will probably involve detailed examination of exactly which objects each transaction reads and writes.

Figure 9–25 represents the state history of the entire system in serialization order, but the slightly different representation of figure 9–26 makes that state history more explicit. In figure 9–26 it appears that each transaction has perversely created a new version of every



value of object at end of transaction

Object	1	2	3	4	5	6	7
A	0	+10	+10	+12	+12	0	0
B	0	−10	−6	−6	−12	−2	−2
C	0	0	−4	−4	+2	+2	+2
D	0	0	0	−2	−2	−2	−2
OUTCOME record state	Committed	Committed	Committed	Aborted	Committed	Pending	Pending

Figure 9–26: System state history with unchanged values shown.

object, with unchanged values in dotted boxes for those objects it did not actually change. This representation emphasizes that the vertical slot for, say, transaction 3 is in effect a reservation in the state history for every object in the system; transaction 3 has an opportunity to propose a new value for any object, if it so wishes.

The reason that the system state history is helpful to the discussion is that as long as we eventually end up with a state history that has the values in the boxes as shown, the actual order in real time in which individual object values are placed in those boxes is unimportant. For example, in figure 9–26, transaction 3 could create its new version of object *C* before transaction 2 creates its new version of *B*. We don't care when things happen, as long as the result is to fill in the history with the same set of values that would result from strictly following this serial ordering. Making the actual time sequence unimportant is exactly our goal, since that allows us to put concurrent threads to work on the various transactions. There are, of course, constraints on time ordering, but they become evident by examining the state history.

Figure 9–26 allows us to see just what time constraints must be observed in order for the system state history to record this particular sequence of transactions. In order for a transaction to generate results appropriate for its position in the sequence, it should use as its input values the latest versions of all of its inputs. If figure 9–26 were available, transaction 4 could scan back along the histories of its inputs *A* and *D*, to the most recent solid boxes (the ones created by transactions 2 and 1, respectively) and correctly conclude that if transactions 2 and 1 have committed then transaction 4 can proceed—even if transaction 3 hasn't gotten around to filling in values for *B* and *C* and hasn't decided whether or not it should commit.

This observation suggests that any transaction has enough information to isolate itself from other transactions correctly if it can discover the dotted-versus-solid status of those version history boxes to its left. The observation also leads to a specific isolation discipline that will ensure correctness. We call this discipline *mark-point* isolation.

2. *The mark-point isolation discipline*

Concurrent threads that invoke READ_CURRENT_VALUE as implemented in figure 9–15 on page 9–42 can not see a pending version of any variable. That observation is useful in designing an isolation discipline, because it allows a transaction to reveal all of its results at once simply by changing the value of its OUTCOME record to COMMITTED. But in addition to that we need a way for later transactions that need to read a pending version to wait for it to become committed. The way to do that is to modify READ_CURRENT_VALUE to wait for, rather than skip over, pending versions created by transactions that are earlier in the sequential ordering (that is, they have a smaller *caller_id*), as implemented in lines 6-11 of figure 9–27. Because, with concurrency, a transaction later in the ordering may create a new version of the same variable before this transaction reads it, READ_CURRENT_VALUE still skips over any versions created by transactions that have a larger *caller_id*. Also, as before, it may be convenient to have a READ_MY_VALUE procedure (not shown) that returns pending values previously written by the running transaction.

Adding the ability to wait for pending versions in READ_CURRENT_VALUE is the first step; to ensure correct isolation we also need to arrange that all variables that a transaction needs as inputs, but that earlier, not-yet-committed transactions plan to modify, have pending

```
1    procedure READ_CURRENT_VALUE (data_id, this_transaction_id)
2    {
3         starting at end of data_id repeat until beginning
4         {
5              v ← previous version of data_id;
6              last_modifier ← v.action_id;
7              if last_modifier ≥ this_transaction_id then skip v;// Continue backward search
8              wait until (last_modifier.outcome_record.state ≠ PENDING)
9              if (last_modifier.outcome_record.state = COMMITTED)
10                  then return v.state;
11                  else skip v;                    // Resume backward search
12        }
13        signal ("Tried to read an uninitialized variable");
14   }
```

Figure 9–27: READ_CURRENT_VALUE for the mark-point isolation discipline. This form of the procedure skips all versions created by transactions later than the calling transaction, and it waits for a pending version created by an earlier transaction until that earlier transaction commits or aborts.

```
procedure NEW_VERSION (reference data_id, this_transaction_id)
{
    if this_transaction_id.outcome_record.mark_state = MARKED then
        signal (“Tried to create new version after announcing mark point!”)
    append new version v to data_id;
    v.value ← NULL;
    v.action_id ← transaction_id;
}

procedure WRITE_VALUE (reference data_id, new_value, this_transaction_id)
{
    starting at end of data_id repeat until beginning
    {
        v ← previous version of data_id;
        if v.action_id = this_transaction_id
            {v.value ← new_value; return}
    }
    signal (“Tried to write without creating new version!”))
}
```

Figure 9–28: Mark-point isolation versions of NEW_VERSION and WRITE_VALUE.

versions. To do that we call on the application programmer (for example, the programmer of the TRANSFER transaction) do a bit of extra work: each transaction should create new, pending versions of every variable it intends to modify, and announce when it is finished doing so. Creating a pending version has the effect of marking those variables that are not ready for reading by later transactions, so we shall call the point at which a transaction has created them all the *mark point* of the transaction. The transaction announces that it has passed its mark point by calling a procedure named MARK_POINT_ANNOUNCE, which simply sets a flag in the outcome record for that transaction.

The mark-point isolation discipline then is that no transaction can begin reading its inputs until the preceding transaction has reached its mark point or is no longer pending. This discipline requires that each transaction identify which data it will update. If the transaction has to modify some data objects before it can discover the identity of others that require update, it could either delay setting its mark point until it does know all of the objects it will write (which would, of course, also delay all succeeding transactions) or use the more complex discipline described in the next section.

For example, in figure 9–26, the boxes under newly arrived transaction 7 are all dotted; transaction 7 should begin by marking the ones that it plans to make solid. For convenience in marking, we split the WRITE_NEW_VALUE procedure of figure 9–15 into two parts, named NEW_VERSION and WRITE_VALUE, as in figure 9–28. Marking then consists simply of a series of calls to NEW_VERSION. When finished marking, the transaction calls MARK_POINT_ANNOUNCE. It may then go about its business, reading and writing values as appropriate to its purpose.

Finally, we enforce the mark point discipline by putting a test and, depending on its outcome, a wait in BEGIN_TRANSACTION, as in figure 9–29, so that no transaction may begin

```
procedure BEGIN_TRANSACTION ()
{
    id ← NEW_OUTCOME_RECORD (PENDING);
    previous_id ← id - 1;
    wait until (previous_id.outcome_record.mark_state = MARKED)
        or (previous_id.outcome_record.state ≠ PENDING);
    return id;
}

procedure NEW_OUTCOME_RECORD (starting_state)
{
    ACQUIRE (outcome_record_lock);                // Isolate this entire procedure.
    id ← TICKET (outcome_record_sequencer);
    allocate id.outcome_record;
    id.outcome_record.state ← starting_state;
    id.outcome_record.mark_state ← NULL;
    RELEASE (outcome_record_lock);
    return id;
}

procedure MARK_POINT_ANNOUNCE (reference this_transaction_id)
{
    this_transaction_id.outcome_record.mark_state ← MARKED;
}
```

Figure 9–29: The procedures BEGIN_TRANSACTION, NEW_OUTCOME_RECORD, and MARK_POINT_ANNOUNCE for the mark-point isolation discipline. BEGIN_TRANSACTION presumes that there is always a preceding transaction. so the system should be initialized by calling NEW_OUTCOME_RECORD to create an empty initial transaction in the *starting_state* COMMITTED and immediately calling MARK_POINT_ANNOUNCE for the empty transaction.

execution until the preceding transaction either reports that it has reached its mark point or is no longer PENDING. Figure 9–29 also illustrates an implementation of MARK_POINT_ANNOUNCE. No changes are needed in procedures ABORT and COMMIT as shown in figure 9–13 on page 9–41, so they are not repeated here.

Because no transaction can start until the previous transaction reaches its mark point, all transactions earlier in the serial ordering must also have passed their mark points, so every transaction earlier in the serial ordering has already created all of the versions that it ever will. Since READ_CURRENT_VALUE now waits for earlier, pending values to become committed or aborted, it will always return to its client a value that represents the final outcome of all preceding transactions. All input values to a transaction thus contain the committed result of all transactions that appear earlier in the serial ordering, just as if it had followed the simple serialization isolation discipline. The result is thus guaranteed to be exactly the same as one produced by a serial ordering, no matter in what real time order the various transactions actually write data values into their version slots. The particular serial ordering that results from this discipline is, as in the case of the simple serialization discipline, the ordering in which the transactions were assigned serial numbers by NEW_OUTCOME_RECORD.

There is one potential interaction between all-or-nothing atomicity and isolation atomicity. If pending versions survive system crashes, at restart the system must track down all PENDING transaction records and mark them ABORTED to ensure that future invokers of READ_CURRENT_VALUE do not wait for the completion of transactions that have forever disappeared.

The mark-point discipline provides isolation by bootstrapping from a more primitive isolation mechanism. As usual in bootstrapping, the idea is to reduce some general problem—here, that problem is to isolate arbitrary application programs—to a special case that is amenable to a special-case solution—here, the special case is construction and initialization of a new outcome record. The procedure NEW_OUTCOME_RECORD in figure 9–29 must itself be isolated, because it may be invoked by several different threads at about the same time and it must be careful to give out different serial numbers to each of them. It must also create completely initialized outcome records, with *value* and *mark_state* set to PENDING and NULL, respectively, because a concurrent thread may immediately need to look at one of those fields. To achieve isolation, NEW_OUTCOME_RECORD bootstraps from the TICKET procedure on page 5–86 to obtain the next sequential serial number, and it uses ACQUIRE and RELEASE to isolate its initialize steps. Those procedures in turn bootstrap from still lower-level isolation mechanisms, so we have three layers of bootstrapping.

We can now reprogram the funds TRANSFER procedure of figure 9–15 to be atomic under both failure and concurrent activity, as in figure 9–30. The major change from the earlier version is addition of lines 4 through 6, in which TRANSFER calls NEW_VERSION to mark the two variables that it intends to modify and then calls MARK_POINT_ANNOUNCE. The interesting

```
1    procedure TRANSFER (reference debit_account, reference credit_account, amount)
2    {
3        my_id ← BEGIN_TRANSACTION ();
4        NEW_VERSION (debit_account, my_id);
5        NEW_VERSION (credit_account, my_id);
6        MARK_POINT_ANNOUNCE (my_id);
7        xvalue ← READ_CURRENT_VALUE (debit_account, my_id);
8        xvalue ← xvalue - amount;
9        WRITE_VALUE (debit_account, xvalue, my_id);
10       yvalue ← READ_CURRENT_VALUE (credit_account, my_id);
11       yvalue ← yvalue + amount;
12       WRITE_VALUE (credit_account, yvalue, my_id);
13       if xvalue > 0 then
14           COMMIT (my_id);
15       else
16       {
17           ABORT (my_id);
18           signal ("Negative transfers are not allowed.");
19       }
20   }
```

Figure 9–30: An implementation of the funds transfer procedure that uses mark point isolation to ensure that it is atomic both with respect to failure and with respect to concurrent activity.

observation about this program is that most of the work of isolation is actually carried out in the called procedures. The only effort or thought required of the application programmer is to identify and mark, by creating new versions, the variables that the transaction will modify.

The delays (which under the simple serialization discipline would all be concentrated in BEGIN_TRANSACTION) are distributed under the mark-point isolation discipline. Some delays may still occur in BEGIN_TRANSACTION, waiting for the preceding transaction to reach its mark point. But if marking is done before any other calculations, transactions are likely to reach their mark points promptly, and thus this delay should be not as great as waiting for them to commit or abort. Delays can also occur at any invocation of READ_CURRENT_VALUE, but only if there is really something that the transaction must wait for, such as committing a pending version of a necessary input variable. Thus the overall delay for any given transaction should never be more than that imposed by the simple serialization discipline, and one might anticipate that it will often be less.

A useful property of the mark-point isolation discipline is that it never creates deadlocks. Whenever a wait occurs it is a wait for some transaction *earlier* in the serialization. That transaction may in turn be waiting for a still earlier transaction, but since no one ever waits for a transaction later in the ordering, progress is guaranteed. The reason is that at all times there must be some earliest pending transaction. The ordering property guarantees that this earliest pending transaction will encounter no waits for isolation, so it, at least, can make progress. When it completes, some other transaction in the ordering becomes earliest, and it now can make progress. Eventually, by this argument, every transaction will be able to make progress. This kind of reasoning about progress is a helpful element of an isolation discipline. In section E of this chapter we shall encounter isolation disciplines that are correct in the sense that they guarantee the same result as a serial ordering, but they do not guarantee progress. Such disciplines require additional mechanisms to assure that threads do not end up deadlocked, waiting for one another forever.

Two other minor points are worth noting. First, if transactions wait to announce their mark point until they are ready to commit or abort, the mark-point isolation discipline reduces to the simple serialization isolation discipline. That observation confirms that one discipline is a relaxed version of the other. Second, there are at least two opportunities in the mark-point isolation discipline to discover and report isolation protocol errors to clients. A transaction should never call NEW_VERSION after announcing its mark point. Similarly, WRITE_VALUE can report an error if the client tries to write a value for which a new version was never created. Both of these error-reporting opportunities are implemented in the pseudocode of figure 9–28.

3. An optimistic isolation discipline: read-capture (advanced topic)

Both simple serialization and mark-point isolation are concurrency control methods that may be described as *pessimistic*. That means that they presume that interference between concurrent transactions is likely and they actively prevent any possibility of interference by imposing waits at any point where interference might occur. An alternative scheme, called *optimistic* concurrency control, is to presume that interference between concurrent transactions is unlikely, and allow them to proceed without waiting. Then, watch for actual interference, and if it happens take some recovery action, for example aborting transactions that do interfere and making them restart. (There is a popular tongue-in-cheek characterization of the difference: pessimistic = "ask first", optimistic = "apologize later".) The goal of optimistic concurrency control is to increase concurrency in situations where actual interference is rare.

The system state history of figure 9–26 suggests an opportunity to be optimistic. We could allow transactions to write values into the system state history in any order and at any time, but with the risk that some attempts to write may be met with the response "Sorry, that write would interfere with another transaction. You must abort, abandon this serialization position in the system state history, obtain a later serialization, and retry your transaction from the beginning."

A specific example of this approach is the *read-capture* discipline. Under the read-capture discipline, there is an option, but not a requirement, of advance marking. Eliminating the requirement of advance marking has the advantage that a transaction does not need to predict the identity of every object it will update—it can discover the identity of those objects as it works. Instead of advance marking, whenever a transaction calls READ_CURRENT_VALUE, that procedure makes a mark at this thread's position in the version history of the object it read. This mark tells potential version-inserters earlier in the serial ordering but arriving later in real time that they are no longer allowed to insert—they must abort and try again, using a later serial position in the version history. Had the prospective version inserter gotten there sooner, before the reader had left its mark, the new version would have been acceptable, and the reader would have instead waited for the version inserter to commit, and taken that new value instead of the earlier one. Read-capture gives the reader the power of extending validity of a version through intervening transactions, up to the reader's own serialization position. This view of the situation is illustrated in figure 9–31, which has the same version history as did figure 9–26.

The key property of read-capture is illustrated by an example in figure 9–31. Transaction 4 was late in creating a new version of object *A*; by the time it tried to do the insertion, transaction 6 had already read the old value (+10) and thereby extended the validity of that old value to the beginning of transaction 6. Therefore, transaction 4 had to be aborted; it has been reincarnated to try again as transaction 7. In its new position as transaction 7, its first act is to read object *D*, extending the validity of its most recent committed value (zero) to the beginning of transaction 7. When it tries to read object *A*, it discovers that the most recent version is still uncommitted, so it must wait for transaction 6 to either commit or abort. Note that if transaction 6 should now decide to create a new version of object *C*, it can do so without any problem, but if it should try to create a new version of object *D*, it would run into a conflict with the old, now extended version of *D*, and it would have to abort.

Read-capture is relatively easy to implement in a version history system. We start, as shown in figure 9–32, by adding a new step (at line 12) to READ_CURRENT_VALUE. This new step records with each data object a *high-water mark*—the serial number of the highest-numbered transaction that has ever read a value from this object's version history. The high-water mark serves as a warning to other transactions that have earlier serial numbers but are late in creating new versions. The warning is that someone later in the serial ordering has already read a version of this object from earlier in the ordering, so it is too late to create a new version now. We guarantee that the warning is heeded by adding a step to NEW_VERSION (at line 23), which checks the high-water mark for the object to be written, to see if any transaction with

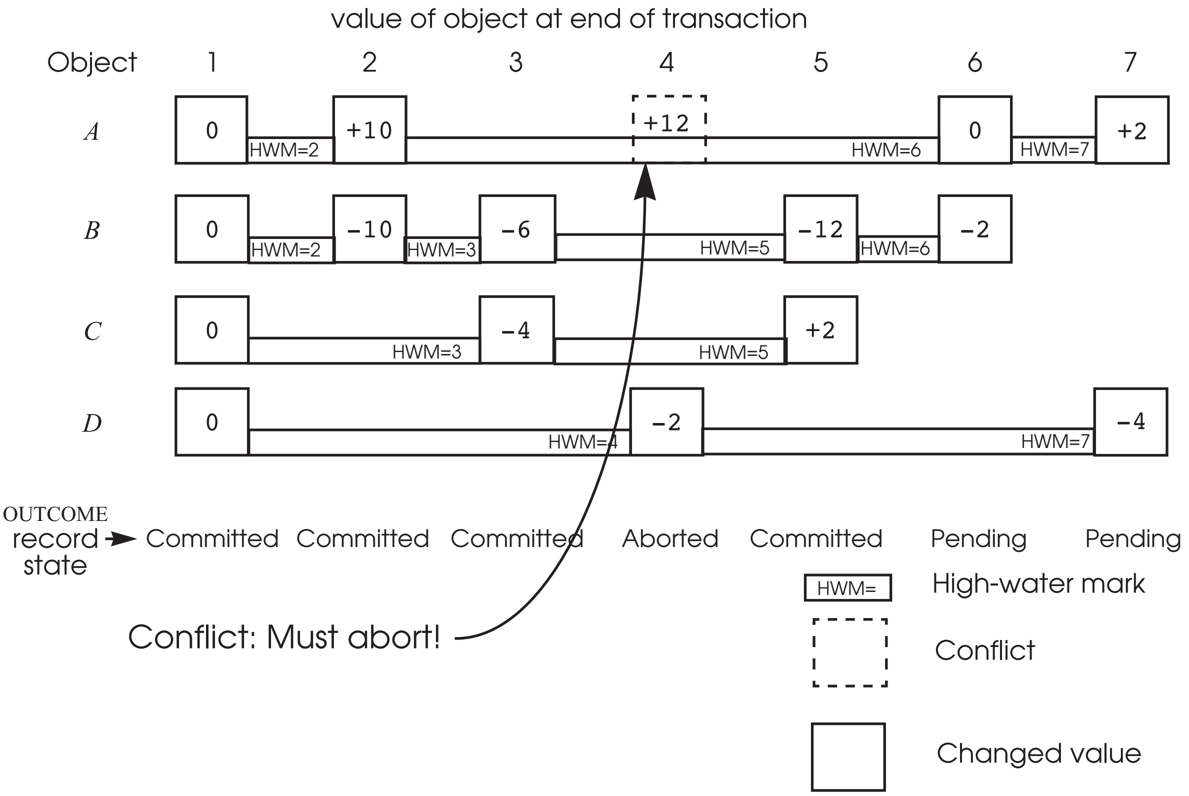


Figure 9–31: Version history with high-water marks and the read-capture discipline. First, transaction 6, which is running concurrently with transaction 4, reads variable *A*, thus extending the high-water mark of *A* to 6. Then, transaction 4 (which intends to transfer 2 from *D* to *A*) encounters a conflict when it tries to create a new version of *A* and discovers that the high-water mark of *A* has already been set by transaction 6, so 4 aborts and returns as transaction 7. Transaction 7 retries transaction 4, extending the high-water marks of *A* and *D* to 7.

a higher serial number has already read the current version of the object. If not, we can create a new version without concern. But if the transaction serial number in the high-water mark is greater than this transaction's own serial number, this transaction must abort, obtain a new, higher serial number, and start over again.

We have removed all constraints on the real-time sequence of the constituent steps of the concurrent transaction, so there is a possibility that a high-numbered transaction will create a new version of some object, and then later a low-numbered transaction will try to create a new version of the same object. Since our NEW_VERSION procedure simply tacks new versions on the end of the object history, we could end up with a history in the wrong order. The simplest way to avoid that mistake is to put an additional test in NEW_VERSION (at line 24), to ensure that every new version has a client serial number that is larger than the serial number of the next previous version. If not, NEW_VERSION aborts the transaction, just as if a read-capture conflict had occurred. (This test aborts only those transactions that perform conflicting *blind writes*, which are uncommon. If either of the conflicting transactions reads

```
procedure READ_CURRENT_VALUE (reference data_id, value, caller_id)
{
    starting at end of data_id repeat until beginning
    {
        v ← previous version of data_id;
        if v.action_id ≥ caller_id then skip v;
        examine v.action_id.outcome_record;
            if PENDING then
                WAIT for v.action_id to COMMIT or ABORT;
                if COMMITTED then
                {
                    v.high_water_mark ← max(v.high_water_mark,
                                                            caller_id);
                    return v.value;
                }
                else skip v;            // Continue backward search
    }
    signal ("Tried to read an uninitialized variable!");
}

procedure NEW_VERSION (reference data_id, caller_id)
{
    if (caller_id < data_id.high_water_mark)            // Conflict with later reader.
    or (caller_id < (LATEST_VERSION[data_id].action_id))     // Blind write conflict.
    then ABORT this transaction and terminate this thread;
    add new version v at end of data_id;
    v.value ← 0;
    v.action_id ← caller_id;
}
procedure WRITE_VALUE (reference data_id, new_value, caller_id)
{
    locate version v of data_id.history such that v.action_id = caller_id;
            (if not found, signal ("Tried to write without creating new version!"))
    v.value ← new_value;
}
```

Figure 9–32: Read-capture forms of READ_CURRENT_VALUE, NEW_VERSION, and WRITE_VALUE.

the value before writing it, the setting and testing of *high_water_mark* will catch and prevent the conflict.)

The first question one must raise about this kind of algorithm is if it actually works: is the result always the same as some serial ordering of the concurrent transactions? Because the read-capture isolation discipline permits greater concurrency than does mark-point, the correctness argument is a bit more involved. The induction part of the argument goes as follows:

1. The WAIT for PENDING values in READ_CURRENT_VALUE ensures that if any pending transaction $k < n$ has modified any value that is later read by transaction n, transaction n will wait for transaction k to commit or abort.

2. The setting of the high-water mark when transaction *n* calls READ_CURRENT_VALUE, together with the test of the high-water mark in NEW_VERSION ensures that if any transaction $j < n$ tries to modify any value after transaction *n* has read that value, transaction *j* will abort and not modify that value.

3. Therefore, every value that READ_CURRENT_VALUE returns to transaction *n* will include the final effect of all preceding transactions $1 \ldots n - 1$.

4. Therefore, every transaction *n* will act as if it serially follows transaction $n - 1$.

Optimistic coordination disciplines such as read-capture have the possibly surprising effect that something done by a transaction later in the serial ordering can cause a transaction earlier in the ordering to abort. This effect is the price of optimism; to be a good candidate for an optimistic discipline, an application probably should not have a lot of data interference.

A subtlety of read-capture is that it is necessary to implement bootstrapping isolation in the procedure NEW_VERSION, by adding a lock and calls to ACQUIRE and RELEASE, because NEW_VERSION can now be called by two concurrent threads that happen to add new versions to the same variable at about the same time. In addition, NEW_VERSION must be careful to keep versions of the same variable in transaction order, so that the backward search performed by READ_CURRENT_VALUE works correctly.

There is one final detail, an interaction with all-or-nothing recovery. High water marks should be stored in volatile memory, so that following a crash (which has the effect of aborting all pending transactions) the high water marks automatically disappear and thus don't cause unnecessary aborts

4. Does anyone actually use version histories for isolation atomicity?

The answer is yes, but the most common use is in an application not likely to be encountered by a software specialist. Legacy processor architectures typically provide a limited number of registers (the "architectural registers") in which the programmer can hold temporary results, but modern large scale integration technology allows space on a physical chip for many more physical registers than the architecture calls for. More registers generally allow better performance, especially in multiple-issue processor designs, which execute several sequential instructions concurrently whenever possible. To allow use of the many physical registers, a register mapping scheme known as *register renaming* implements a version history for the architectural registers. This version history allows instructions that would interfere with each other only because of a shortage of registers to execute concurrently.

For example, Intel Pentium processors, which are based on the x86 instruction set architecture described in appendix 5–A on page 5–91, have only eight architectural registers. The Pentium 4 has 128 physical registers, and a register renaming scheme based on a circular *reorder buffer*. A reorder buffer resembles a direct hardware implementation of the procedures NEW_VERSION and WRITE_VALUE of figure 9–28 on page 9–66. As each instruction

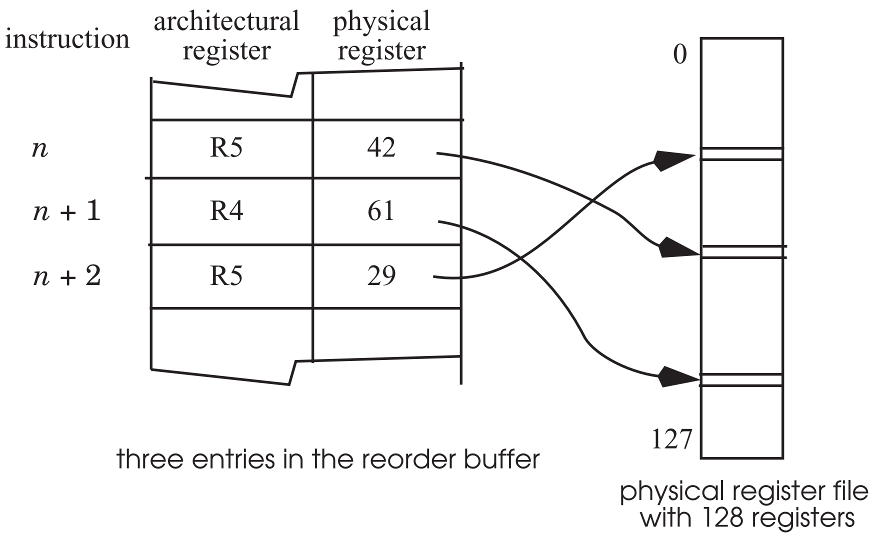


Figure 9–33: Example showing how a reorder buffer maps architectural register numbers to physical register numbers. The program sequence corresponding to the three entries is:

```
n          R5 ← R4 × R2            // Write a result in register five.
n + 1      R4 ← R5 + R1            // Use result in register five.
n + 2      R5 ← READ (117492)      // Write content of a memory cell in register five.
```

Instructions *n* and *n* + 2 both write into register R5, so R5 has two versions, with mappings to physical registers 42 and 29, respectively. Instruction *n* + 2 can thus execute concurrently with instructions *n* and *n* + 1.

issues (which corresponds to BEGIN_TRANSACTION), it is assigned the next sequential slot in the reorder buffer. The slot is a map that holds two numbers: the number of the architectural register that the programmer specified to hold the output value of the instruction, and the number of one of the 128 physical registers, the one that will actually hold that output value. Since machine instructions have just one output value, assigning a slot in the reorder buffer implements in a single step the effect of both NEW_OUTCOME_RECORD and NEW_VERSION. Similarly, when the instruction commits, it places its output in that physical register, thereby implementing WRITE_VALUE and COMMIT as a single step.

Figure 9–33 illustrates register renaming with a reorder buffer. In the program sequence of that example, instruction *n* uses architectural register five to hold an output value that instruction *n* + 1 will use as an input. Instruction *n* + 2 loads architectural register five from memory. Register renaming allows there to be two (or more) versions of register five simultaneously, one version (in physical register 42) containing a value for use by instructions *n* and *n* + 1 and the second version (in physical register 29) to be used by instruction *n* + 2. The performance benefit is that instruction *n* + 2 (and any later instructions that write into architectural register 5) can proceed concurrently with instructions *n* and *n* + 1. An instruction following instruction *n* + 2 that requires the new value in architectural register five as an input uses a hardware implementation of READ_CURRENT_VALUE to locate the most recent preceding mapping of architectural register five in the reorder buffer. In this case that most recent mapping is to physical register 29. The later instruction then stalls, waiting for instruction $n + 2$ to write a value into physical register 29. Later instructions that reuse architectural register five for some purpose that does not require that version can proceed concurrently.

Although register renaming is conceptually straightforward, the mechanisms that prevent interference when there are dependencies between instructions tend to be more intricate than either mark-point or read-capture isolation, so this description has been oversimplified. For more detail, the reader should consult a textbook on processor architecture, for example *Computer Architecture, a Quantitative Approach*, by Hennessy and Patterson [Suggestions for Further Reading 1.1.2].

The Oracle database management system offers several isolation methods, one of which it calls "serializable", though the label may be a bit misleading. This method uses an isolation scheme that the database literature calls *snapshot isolation*. The idea is that when a transaction begins the system conceptually takes a snapshot of every committed value and the transaction reads all of its inputs from that snapshot. If two concurrent transactions (which might start with the same snapshot) modify the same variable, the first one to commit wins; the system aborts the other one with a "serialization error". This scheme effectively creates a limited variant of a version history that, in certain situations, does not correctly isolate concurrent transactions. Other software implementations of version histories for isolation have been explored primarily in research environments. Designers of database systems usually use locks rather than version histories, because there is more experience in achieving high performance with locks. Isolation by the systematic use of locks is the subject of the next section of this chapter.

E. Isolation atomicity II: Pragmatics

The previous section showed that a version history system that provides all-or-nothing atomicity can be extended to also provide isolation atomicity. When the all-or-nothing atomicity design uses a log and installs data updates in cell storage, other, concurrent actions can again immediately see those updates, so we again need a scheme to provide isolation atomicity. When a system uses logs for all-or-nothing atomicity, it usually adopts the mechanism introduced in chapter 5—*locks*—for isolation atomicity. However, as chapter 5 pointed out, programming with locks is hazardous, and the traditional programming technique of debugging until the answers seem to be correct is unlikely to catch all locking errors. We now revisit locks, this time with the goal of using them in stylized ways that allow us to develop arguments that the locks correctly implement isolation.

1. Locks

To review, a *lock* is a flag associated with a data object and set by an action to warn other, concurrent, actions not to read or write the object. Conventionally, a locking scheme involves two procedures:

```
ACQUIRE (A.lock);
```

marks a lock variable associated with object *A* as having been acquired. If the object is already acquired, ACQUIRE waits until the previous acquirer releases it.

```
RELEASE (A.lock);
```

unmarks the lock variable associated with *A*, perhaps ending some other action's wait for that lock. For the moment, we assume that the semantics of a lock follow the single-acquire protocol of chapter 5: if two or more actions attempt to acquire a lock at about the same time, only one shall succeed; the others must find the lock already acquired. Later, on page 9–82, we shall consider some alternative protocols, for example one that permits several readers of a variable as long as there is no one writing it.

The biggest problem with locks is that programming errors can lead to actions that are not correctly isolated. Lack of isolation of concurrent actions can in turn lead to races that, because the interfering actions are timing dependent, can make it extremely difficult to figure out what went wrong. Thus a primary goal is that isolation of concurrent transactions should be arguably correct. For locks, the way to achieve this goal is to follow three steps systematically:

- Develop a locking discipline that specifies which locks must be acquired and when.

- Establish a compelling line of reasoning that concurrent transactions that follow the discipline will be correctly isolated.
- Interpose a *lock manager*, a program that enforces the discipline, between the programmer and the ACQUIRE and RELEASE procedures.

Many locking disciplines have been designed and deployed, including some that fail to correctly isolate transactions (for an example, see problem 9.5, on page 14–30). We examine three disciplines that succeed.

The first, and simplest, discipline that does isolate transactions correctly is the *system-wide lock*. When the system first starts operation, it creates a single lockable variable, S, in volatile memory. The discipline is that every transaction must start with

```
ACQUIRE (S.lock);
```

and every transaction must end with

```
RELEASE (S.lock);
```

Enforcing this discipline is straightforward: the BEGIN step of a transaction always acquires *S.lock*, and the END step always releases it, independent of whether the result was COMMIT or ABORT. Any programmer who creates a new transaction then has a guarantee that it is isolated from all other transactions.

The system-wide lock discipline allows only one transaction to execute at a time. It serializes potentially concurrent transactions in the order that they call ACQUIRE. The system-wide lock discipline is in all respects identical to the simple serialization discipline of section D. In fact, the simple serialization pseudocode

```
id ← NEW_OUTCOME_RECORD ();
preceding_id ← id - 1;
wait until preceding_id.outcome_record.value ≠ PENDING;
…
COMMIT (id); [or ABORT (id);]
```

and the system-wide lock invocation

```
ACQUIRE (S.lock);
…
RELEASE (S.lock);
```

are actually just two implementations of the same idea.

As with simple serialization, system-wide locking restricts concurrency in cases where it doesn't need to, because it locks all data touched by every transaction. Thus there is an interest in developing less restrictive locking disciplines. The starting point is usually to employ a finer lock *granularity*: lock smaller objects, such as individual data records, individual pages of data records, or even fields within records. The trade-off in gaining concurrency is that when there is more than one lock, correctness arguments become more complex. Fortunately, there are some disciplines for which correctness arguments are feasible.

The second locking discipline, known as *simple locking,* is similar in spirit to, though not quite identical with, the mark-point isolation discipline. The discipline has two rules. First, each transaction must acquire a lock for every shared data object it intends to read or write before doing any actual reading and writing. Second, it may release its locks only after the transaction installs its last update and commits or completely restores the data and aborts. Analogous to the mark point, the transaction has what is called a *lock point*: the first instant at which it has acquired all of its locks. The collection of locks it has acquired when it reaches its lock point is called its *lock set*. A lock manager enforces simple locking by requiring that each transaction supply its intended lock set as an argument to the atomic BEGIN operation, which acquires all of the locks of the lock set, if necessary waiting for them to become available. The lock manager can also interpose itself on all calls to read data and to log updates, to verify that they refer to variables that are in the lock set. The lock manager also intercepts the call to commit or abort (or, if the application uses roll-forward recovery, to log an END record) at which time it automatically releases all of the locks of the lock set.

The simple locking discipline correctly isolates concurrent transactions. We can make that claim using a line of argument analogous to the one used for correctness of the mark-point isolation discipline. Imagine that an all-seeing outside observer maintains an ordered list to which it adds each transaction identifier as soon as the transaction reaches its lock point and removes it from the list when it begins to release its locks. Under the simple locking discipline each transaction has agreed not to read or write anything until that transaction has been added to the observer's list. We also know that all transactions that precede this one in the list must have already passed their lock point. Since no data object can appear in the lock sets of two transactions, no data object in any transaction's lock set appears in the lock set of the transaction preceding it in the list. Thus all of this transaction's input values are the same as they will be when the preceding transaction in the list commits or aborts. The same argument applies to the transaction before the preceding one, and by induction to every transaction earlier in the list, so all inputs to any transaction are identical to the inputs that would be available if all the transactions ahead of it in the list ran serially, in the order of the list. Thus the simple locking discipline correctly isolates transactions. Concurrent transactions will produce results as if they had been serialized in the order that they reached their lock points.

As with mark-point isolation, simple locking misses some opportunities for concurrency. In addition, the simple locking discipline creates a problem that can be significant in some applications. Because it requires the transaction to acquire a lock on every shared object that it will either read *or* write (recall that mark-point isolation requires marking only of shared objects that the transaction will write), applications that discover which objects need to be read by reading other shared data objects have no alternative but to lock every object that they *might* need to read. To the extent that the set of objects for which an application *might* need to acquire read locks is larger than the set for which it eventually *does* acquire read locks, the simple locking discipline can interfere with opportunities for concurrency.

The third locking discipline, called *two-phase locking,* like the read-capture discipline, avoids the requirement that a transaction know in advance which locks it must acquire. It is widely used, but its correctness is harder to establish. The two-phase locking discipline allows a transaction to acquire locks as it proceeds, and the transaction may read or write a data object as soon as it acquires a lock on that object. The primary constraint is that the transaction may not release any locks until it passes its lock point. Further, the transaction can release a lock on a read-only object any time after it reaches its lock point *if* it will never need that object again, even to abort. The name of the discipline comes about because the number of locks acquired by a transaction monotonically increases up to the lock point (the first phase), after which it monotonically decreases (the second phase). Just as with simple locking, two-phase locking isolates concurrent transactions so that they produce results as if they had been serialized in the order they reach their lock points. A lock manager can implement two-phase locking by intercepting all calls to read and write data; it acquires a lock (perhaps having to wait) on the first use of each shared variable. As with simple locking, it then holds the locks until it intercepts the call to commit, abort, or log the END record of the transaction, at which time it releases them all at once.

The extra flexibility of two-phase locking makes it harder to argue that it guarantees correct isolation. Informally, once a transaction has acquired a lock on a data object, the value of that object is the same as it will be when the transaction reaches its lock point, so reading that value now must yield the same result as waiting till then to read it. Furthermore, releasing a lock on an object that it hasn't modified must be harmless if this transaction will never look at the object again, even to abort. A formal argument that two-phase locking leads to correct isolation can be found in most advanced texts on concurrency control and transactions. See, for example, *Transaction Processing*, by Gray and Reuter [Suggestions for Further Reading 1.1.5].

There are two interactions between locks and logs that require some thought: individual transactions that abort, and system recovery. Aborts are the easiest to deal with. Since we require that an aborting transaction restore its changed data objects to their original values before releasing any locks, no special account need be taken of aborted transactions. From an isolation point of view they look just like committed transactions that didn't change anything. The rule about not releasing any locks on modified data before the end of the transaction is essential to accomplishing an abort. If a lock on some modified object were released, and then the transaction decided to abort, it might find that some other transaction has now acquired that lock and changed the object again. Backing out an aborted change is likely to be impossible unless the locks on modified objects have been held.

The interaction between log-based recovery and locks is less obvious. The question is whether locks themselves are data objects for which changes should be logged. To analyze this question, suppose there is a system crash. At the completion of crash recovery there should be no pending transactions, because any transactions that were pending at the time of the crash should have been rolled back by the recovery procedure, and recovery does not allow any new transactions to begin until it completes. Since locks exist only to isolate pending transactions, it would clearly be an error if there were locks still set when crash recovery is complete. That observation suggests that locks belong in volatile storage, where they will automatically disappear on a crash, rather than in non-volatile storage, where the recovery procedure would have to hunt them down to release them. The bigger question, however, is whether or not the log-based recovery algorithm will construct a correct system state—correct in the sense that it could have arisen from some serial ordering of those transactions that committed before the crash.

Continue to assume that the locks are in volatile memory, and at the instant of a crash all record of the locks is lost. Some set of transactions—the ones that logged a BEGIN record but have not yet logged an END record—may not have been completed. But we know that the transactions that were not complete at the instant of the crash had non-overlapping lock sets at the moment that the lock values vanished. The recovery algorithm of figure 9–22 will systematically UNDO or REDO installs for the incomplete transactions, but every such UNDO or REDO must modify a variable whose lock was in some transaction's lock set at the time of the crash. Because those lock sets were non-overlapping, the correctness of isolation during recovery cannot be affected by the absence of the locks. The locks created a particular serialization of the transactions and the log has captured that serialization. Since RECOVER performs UNDO actions in reverse order as specified in the log, and it performs REDO actions in forward order, again as specified in the log, RECOVER reconstructs exactly that same serialization. Thus even a recovery algorithm that reconstructs the entire data base from the log is guaranteed to produce the same serialization as when the transactions were originally performed. So long as no new transactions begin until recovery is complete, there is no danger of miscoordination, despite the absence of locks during recovery.

2. *Performance optimizations*

Most logging-locking systems are substantially more complex than the description so far might lead one to expect. The complications primarily arise from attempts to gain performance. In section C.6 we saw how buffering of disk I/O in a volatile memory cache, to allow reading, writing, and computation to go on concurrently, can complicate a logging system. Designers sometimes apply two performance-enhancing complexities to locking systems: physical locking and adding lock compatibility modes.

A performance-enhancing technique driven by buffering of disk I/O and physical media considerations is to choose a particular lock granularity known as *physical locking*. If a transaction makes a change to a six-byte object in the middle of a 1000-byte disk sector, or to a 1500-byte object that occupies parts of two disk sectors, there is a question about which "variable" should be locked: the object, or the disk sector(s)? If two concurrent threads make updates to unrelated data objects that happen to be stored in the same disk sector, then the two disk writes must be coordinated. Choosing the right locking granularity can make a big performance difference.

Locking application-defined objects without consideration of their mapping to physical disk sectors is appealing because it is understandable to the application writer. For that reason, it is usually called *logical locking.* In addition, if the objects are small, it apparently allows more concurrency: if another transaction is interested in a different object that is in the same disk sector, it could proceed in parallel. However, a consequence of logical locking is that logging must also be done on the same logical objects. Different parts of the same disk sector may be modified by different transactions that are running concurrently, and if one transaction commits but the other aborts neither the old nor the new disk sector is the correct one to restore following a crash; the log entries must record the old and new values of the individual data objects that are stored in the sector. Finally, recall that a high-performance logging system with a cache must, at commit time, force the log to disk and keep track of which objects in the cache it is safe to write to disk without violating the write-ahead log protocol. So logical locking with small objects can escalate cache record-keeping.

Backing away from the details, high-performance disk management systems typically require that the argument of a PUT call be a block whose size is commensurate with the size of a disk sector. Thus the real impact of logical locking is to create a layer between the application and the disk management system that presents a logical, rather than a physical, interface to its transaction clients; such things as data object management and garbage collection within disk sectors would go into this layer. The alternative is to tailor the logging and locking design to match the native granularity of the disk management system. Since matching the logging and locking granularity to the disk write granularity can reduce the number of disk operations, both logging changes to and locking blocks that correspond to disk sectors rather than individual data objects is a common practice.

Another performance refinement appears in most locking systems: the specification of *lock compatibility modes*. The idea is that when a transaction acquires a lock, it can specify what operation (for example, READ or WRITE) it intends to perform on the locked data item. If that operation is compatible—in the sense that the result of concurrent transactions is the same as some serial ordering of those transactions—then this transaction can be allowed to acquire a lock even though some other transaction has already acquired a lock on that same data object.

The most common example involves replacing the single-acquire locking protocol with the *multiple-reader, single-writer protocol*. According to this protocol, one can allow any number of readers to simultaneously acquire read-mode locks for the same object. The purpose of a read-mode lock is to assure that no other thread can change the data while the lock is held. Since concurrent readers do not present an update threat, it is safe to allow any number of them. If another transaction needs to acquire a write-mode lock for an object on which several threads already hold read-mode locks, that new transaction will have to wait for all of the readers to release their read-mode locks. There are many applications in which a majority of data accesses are for reading, and for those applications the provision of read-mode lock compatibility can reduce the amount of time spent waiting for locks by orders of magnitude. At the same time, the scheme adds complexity, both in the mechanics of locking and also in policy issues, such as what to do if, while a prospective writer is waiting for readers to release their read-mode locks, another thread calls to acquire a read-mode lock. If there is a steady stream of arriving readers, a writer could be delayed indefinitely.

This description of performance optimizations and their complications is merely illustrative, to indicate the range of opportunities and kinds of complexity that they engender; there are many other performance-enhancement techniques, some of which can be effective, and others that are of dubious value; most have different values depending on the application. For example, some locking disciplines compromise isolation by allowing transactions to read data values that are not yet committed. As one might expect, the complexity of reasoning about what can or cannot go wrong in such situations escalates. If a designer intends to implement a system using performance enhancements such as buffering, lock compatibility modes, or compromised isolation, it would be advisable to study carefully the book by Gray and Reuter, as well as existing systems that implement similar enhancements.

3. Deadlock; making progress

Chapter 5 mentioned the problem of *deadlock*, and introduced several ideas surrounding it. Now, we have more tools available to deal with this set of problems, so it is worth revisiting that discussion in its entirety. The possibility of deadlock is an inevitable consequence of using locks to coordinate concurrent activities. Any number of concurrent transactions can get hung up in a deadlock, either waiting for one another, or simply waiting for a lock to be released by some transaction that is already deadlocked. Deadlock leaves us a significant loose end: correctness arguments assure us that any transactions that complete will produce results as though they were run serially, but they say nothing about whether or not any transaction will ever complete. In other words, our system may assure *correctness*, in the sense that no wrong answers ever come out, but it does not assure *progress*—no answers may come out at all.

Chapter 5 suggested lock ordering as one way of preventing deadlock. In practice, techniques for coping with deadlock fall into one of three general categories:

1. *Timer expiration*. When a new transaction begins, the lock manager sets an interrupting timer to a value somewhat greater than the time it should take for the transaction to complete. If a transaction gets into a deadlock, its timer will expire, at which point the system aborts that transaction, rolling back its changes and releasing its locks in the hope that the other transactions involved in the deadlock may be able to proceed. If not, another one will time out, releasing further locks. Timing out deadlocks is effective, though it has the usual defect: it is difficult to choose a suitable timer value that keeps things moving along but also accommodates normal delays and variable operation times. If the environment or system load changes, it may be necessary to readjust all such timer values, an activity that can be a real nuisance in a large system.

2. *Lock ordering*. As suggested in chapter 5, number the locks uniquely, and require that transactions acquire locks in ascending numerical order. With this plan, when a transaction encounters an already-acquired lock, it is always safe to wait for it, since the transaction that previously acquired it cannot be waiting for any locks that this transaction has already acquired—all those locks are lower in number than this one. There is thus a guarantee that somewhere, at least one transaction (the one holding the highest-numbered lock) can always make progress. When that transaction finishes, it will release all of its locks, and some other transaction will become the one that is guaranteed to be able to make progress. A generalization of lock ordering that may eliminate some unnecessary waits is to arrange the locks in a lattice and require that they be acquired in some lattice traversal order. The trouble with lock ordering, as with simple locking, is that some applications may not be able to predict all of the needed locks before setting the first one. Chapter 5 also mentioned an elegant strategy devised by Andre Bensoussan in 1966, in which a transaction may acquire locks in any order, but if it encounters an acquired lock with a number lower than one it has previously acquired itself, the transaction must back up (in terms of this chapter, UNDO previous actions) just far enough to release its higher-numbered locks, wait for the lower-numbered lock to become available, acquire that lock, and then REDO the backed-out actions.

3. *Cycle detection*. Maintain, in the lock manager, a schedule of which transactions have acquired which locks and which transactions are waiting for which locks. Whenever another transaction tries to acquire a lock and finds it is already locked, the lock manager can examine the schedule to see if waiting would produce a cycle of waits, and thus a deadlock. If it would, the lock manager selects some cycle member to be the victim, and unilaterally aborts that transaction, so that the others may continue. The aborted transaction then retries in the hope that the other transactions have made enough progress to be out of the way and another deadlock will not occur.

When a system uses lock ordering or cycle detection, it is common to also set a timer as a safety net, because a hardware failure or a programming error such as an endless loop can create a progress-blocking situation that neither lock ordering nor cycle detection can catch.

Since a deadlock detection algorithm can introduce an extra reason to abort a transaction, one can envision pathological situations where the algorithm aborts every attempt to perform some particular transaction, no matter how many times its invoker retries. Suppose, for example, that two threads named Alphonse and Gaston get into a deadlock trying to acquire locks for two objects named Apple and Banana: Alphonse acquires the lock for Apple, Gaston acquires the lock for Banana, Alphonse tries to acquire the lock for Banana and waits, then Gaston tries to acquire the lock for Apple and waits, creating the deadlock. Eventually, Alphonse times out and begins rolling back updates in preparation for releasing locks. Meanwhile, Gaston times out and does the same thing. Both restart, and they get into another deadlock, with their timers set to expire exactly as before, so they will probably repeat the sequence forever. Thus we still have no guarantee of progress. This is the situation that chapter 5 called *livelock*, since formally there is no deadlock and both threads are busy doing something that looks superficially useful.

One way to deal with livelock is to apply a randomized version of a technique familiar from chapter 7: *exponential random backoff*. When a timer expiration leads to an abort, the lock manager, after clearing the locks, delays that thread for a random length of time, chosen from some starting interval, in the hope that the randomness will change the relative timing of the livelocked transactions enough that on the next try one will succeed and then the other can then proceed without interference. If the transaction again encounters interference, it tries again, but on each retry not only does the lock manager choose a new random delay, but it also increases the interval from which the delay is chosen by some multiplicative constant, typically 2. Since on each retry there is an increased probability of success, one can push this probability as close to unity as desired by continued retries, with the expectation that the interfering transactions will eventually get out of one another's way. A useful property of exponential random backoff is that if repeated retries continue to fail it is almost certainly an indication of some deeper problem—perhaps a programming mistake or a level of competition for shared variables that is intrinsically so high that the system should be redesigned.

The design of more elaborate algorithms or programming disciplines that guarantee progress is a project that has only modest potential payoff, and an *end-to-end argument* suggests that it may not be worth the effort. In practice, systems that would have frequent interference among transactions are not usually designed with a high degree of concurrency anyway. When interference is not frequent, simple techniques such as safety-net timers and exponential random backoff not only work well, but they usually must be provided anyway, to cope with any races or programming errors such as endless loops that may have crept into the system design or implementation. Thus a more complex progress-guaranteeing discipline is likely to be redundant, and only rarely will it get a chance to promote progress.

F. Atomicity across layers and multiple sites

There remain some important gaps in our exploration of atomicity. First, in a layered system, a transaction implemented in one layer may consist of a series of component actions of a lower layer that are themselves atomic. The question is how the commitment of the lower-layer transactions should relate to the commitment of the higher layer transaction. If the higher-layer transaction decides to abort, the question is what to do about lower-layer transactions that may have already committed. There are two possibilities:

- Reverse the effect of any committed lower-layer transactions with an UNDO action. This technique requires that the results of the lower-layer transactions be visible only within the higher-layer transaction.

- Somehow delay commitment of the lower-layer transactions and arrange that they actually commit at the same time that the higher-layer transaction commits.

Up to this point, we have assumed the first possibility. In this section we explore the second one.

Another gap is that, as described so far, our techniques to provide atomicity all involve the use of shared variables in memory or storage (for example, pointers to the latest version, outcome records, logs, and locks) and thus implicitly assume that the composite actions that make up a transaction all occur in close physical proximity. When the composing actions are physically separated, communication delay, communication reliability, and independent failure make atomicity both more important and harder to achieve.

We shall edge up on both of these problems by first identifying a common subproblem: implementing nested transactions. We will then extend the solution to the nested transaction problem to create an agreement protocol, known as *two-phase commit,* that coordinates commitment of lower-layer transactions. We can then extend the two-phase commit protocol, using a specialized form of remote procedure call, to coordinate steps that must be carried out at different places. This sequence is another example of bootstrapping; the special case that we know how to handle is the single-site transaction and the more general problem is the multiple-site transaction. As an additional observation, we will discover that multiple-site transactions are very similar to, but not quite the same as, the *dilemma of the two generals*.

1. Hierarchical composition of transactions

We got into the discussion of transactions by considering that complex interpreters are engineered in layers, and that each layer should implement atomic actions for its next-higher, client layer. Thus transactions are nested, each one typically consisting of multiple lower-layer transactions. This nesting requires that some additional thought be given to the mechanism of achieving atomicity.

Consider again a banking example. Suppose that the TRANSFER procedure of page 9–21 is available for moving funds from one account to another, and it has been implemented as a transaction. Suppose now that we wish to create the two application procedures of figure 9–34. The first procedure, PAY_INTEREST, invokes TRANSFER to move an appropriate amount of money from or to an internal account named *bank*, the direction and rate depending on whether the customer account balance is positive or negative. The second procedure, MONTH_END_INTEREST, fulfills the bank's intention to pay (or extract) interest every month on every customer account by iterating through the accounts and invoking PAY_INTEREST on each one.

```
procedure PAY_INTEREST (reference account)
{
    if account.balance > 0 then
    {
        interest = account.balance * 0.05;
        TRANSFER (bank, account, interest);
    }
    else
    {
        interest = account.balance * 0.15;
        TRANSFER (account, bank, interest);
    }
    return;
}

procedure MONTH_END_INTEREST:()
{
    for A ← each customer_account do
        PAY_INTEREST (A);
    return;
}
```

Figure 9–34: An example of two procedures, one of which calls the other, yet each should be individually atomic.

It would probably be inappropriate to have two invocations of MONTH_END_INTEREST running at the same time, but it is likely that at the same time that MONTH_END_INTEREST is running there are other banking activities in progress that are also invoking TRANSFER. It is also possible that the **for each** statement inside MONTH_END_INTEREST actually runs several instances of its iteration (and thus of PAY_INTEREST) concurrently. Thus we have a need for three layers of transactions. The lowest layer is the TRANSFER procedure, in which debiting of one account and crediting of a second account must be atomic. At the next higher layer, the procedure PAY_INTEREST should be executed atomically, to ensure that some concurrent TRANSFER transaction doesn't change the balance of the account between the positive/negative test and the calculation of the interest amount. Finally, the procedure MONTH_END_INTEREST should be a transaction, to ensure that some concurrent TRANSFER transaction does not move money from an account A to an account B between the interest-payment processing of those two accounts, since such a transfer could cause the bank to pay interest twice on the same funds. Structurally, an invocation of the TRANSFER procedure is nested inside PAY_INTEREST, and one or more concurrent invocations of PAY_INTEREST are nested inside MONTH_END_INTEREST.

The reason nesting is a potential problem comes from a consideration of the commit steps of the nested transactions. For example, the commit point of the TRANSFER transaction would seem to have to occur either before or after the commit point of the PAY_INTEREST transaction, depending on where in the programming of PAY_INTEREST we place its commit point. Yet either of these positions will cause trouble. If the TRANSFER commit occurs in the pre-commit phase of PAY_INTEREST then if there is a system crash PAY_INTEREST will not be able to back out as though it hadn't tried to operate because the values of the two accounts that TRANSFER changed may have already been used by concurrent transactions to make payment decisions. But if the TRANSFER commit does not occur until the post-commit phase of PAY_INTEREST, there is a risk that the transfer itself can not be completed, for example because one of the accounts is inaccessible. The conclusion is that somehow the commit point of the nested transaction should coincide with the commit point of the enclosing transaction. A slightly different coordination problem applies to MONTH_END_INTEREST: no TRANSFERs by other transactions should occur while it runs (that is, it should be isolated from concurrent TRANSFER transactions), but it must be able to do multiple TRANSFERs itself, each time it invokes PAY_INTEREST, and its own possibly concurrent transfers must be isolated from one another, since they all involve the account named "bank".

Suppose for the moment that the system provides transactions with version histories. We can deal with nesting problems by extending the idea of an outcome record: we allow outcome records to be organized hierarchically. Whenever we create a nested transaction, we record in its outcome record both the initial state (PENDING) of the new transaction and the identifier of the enclosing transaction. The resulting hierarchical arrangement of outcome records then exactly reflects the nesting of the transactions. A top-layer outcome record would contain a flag to indicate that it is not nested inside any other transaction. When an outcome record contains the identifier of a higher-layer transaction, we refer to it as a *dependent* outcome record, and the record to which it refers is called its *superior*.

The transactions, whether nested or enclosing, then go about their business, and depending on their success mark their own outcome records COMMITTED or ABORTED, as usual. However, when READ_CURRENT_VALUE (described on page 9–65) examines the status of a version to see whether or not the transaction that created it is COMMITTED, it must additionally check to see if the outcome record contains a reference to a superior outcome record. If so, it must follow the reference and check the status of the superior. If that record says that it, too, is COMMITTED, it must continue following the chain upward, if necessary all the way to the highest-layer outcome record. The transaction in question is actually COMMITTED only if all the records in the chain are in the COMMITTED state. If any record in the chain is ABORTED, this transaction is actually ABORTED, despite the COMMITTED claim in its own outcome record. Finally, if neither of those situations holds, then there must be one or more records in the chain that are still PENDING. The outcome of this transaction remains PENDING until those records become COMMITTED or ABORTED. Thus the outcome of an apparently-COMMITTED dependent outcome record actually depends on the outcomes of all of its ancestors. We can describe this situation by saying that, until all its ancestors commit, this lower-layer transaction is sitting on a knife-edge, at the point of committing but still capable of aborting if necessary. For purposes of discussion we shall identify this situation as a distinct virtual state of the outcome record and the transaction, by saying that the transaction is *tentatively committed*.

This hierarchical arrangement has several interesting programming consequences. If a nested transaction has any post-commit steps, those steps cannot proceed until all of the hierarchically higher transactions have committed. For example, if one of the nested transactions opens a cash drawer when it commits, the sending of the release message to the cash drawer must somehow be held up until the highest-layer transaction determines its outcome.

This output visibility consequence is only one example of many relating to the tentatively committed state. The nested transaction, having declared itself tentatively committed, has renounced the ability to abort—the decision is in someone else's hands. It must be able to run to completion *or* to abort, and it must be able to maintain the tentatively committed state indefinitely. Maintaining the ability to go either way can be awkward, since the transaction may be holding locks, keeping pages in memory or tapes mounted, or reliably holding on to output messages. One consequence is that a designer cannot simply take any arbitrary transaction and blindly use it as a nested component of a larger transaction. At the very least, the designer must review what is required for the nested transaction to maintain the tentatively committed state.

Another, more complex, consequence arises when one considers possible interactions among different transactions that are nested within the same higher-layer transaction. Consider our earlier example of TRANSFER transactions that are nested inside PAY_INTEREST, which in turn is nested inside MONTH_END_INTEREST. Suppose that the first time that MONTH_END_INTEREST invokes PAY_INTEREST, that invocation commits, thus moving into the tentatively committed state, pending the outcome of MONTH_END_INTEREST. Then MONTH_END_INTEREST invokes PAY_INTEREST on a second bank account. PAY_INTEREST needs to be able to read as input data the value of the bank's own interest account, which is a pending result of the previous, tentatively COMMITTED, invocation of PAY_INTEREST. The READ_CURRENT_VALUE algorithm, as implemented on page 9–65, doesn't distinguish between reads arising within the same group of nested transactions and reads from some completely unrelated transaction. Figure 9–35 illustrates the situation. If the test in READ_CURRENT_VALUE for committed values is extended by simply following the ancestry of the outcome record controlling the latest version, it will undoubtedly force the second invocation of PAY_INTEREST to wait pending the final outcome of the first invocation of PAY_INTEREST. But since the outcome of that first invocation depends on the outcome of MONTH_END_INTEREST, and the outcome of MONTH_END_INTEREST currently depends on the success of the second invocation of PAY_INTEREST, we have a built-in cycle of waits that at best can only time out and abort.

Since blocking the read would be a mistake, the question of when it might be OK to permit reading of data values created by tentatively COMMITTED transactions requires some further thought. The isolation requirement is that no update made by a tentatively COMMITTED transaction should be visible to any transaction that would survive if for some reason the tentatively COMMITTED transaction ultimately aborts. Within that constraint, updates of tentatively COMMITTED transactions can freely be passed around. We can achieve that goal in the following way: compare the outcome record ancestry of the transaction doing the read with the ancestry of the outcome record that controls the version to be read. If these ancestries do not merge (that is, there is no common ancestor) then the reader must wait for the version's ancestry to be completely committed. If they do merge and all the transactions in the ancestry of the data version that are below the point of the merge are tentatively committed, no wait is necessary. Thus, in figure 9–35, MONTH_END_INTEREST might be running the two (or more) invocations of PAY_INTEREST concurrently. Each invocation will call

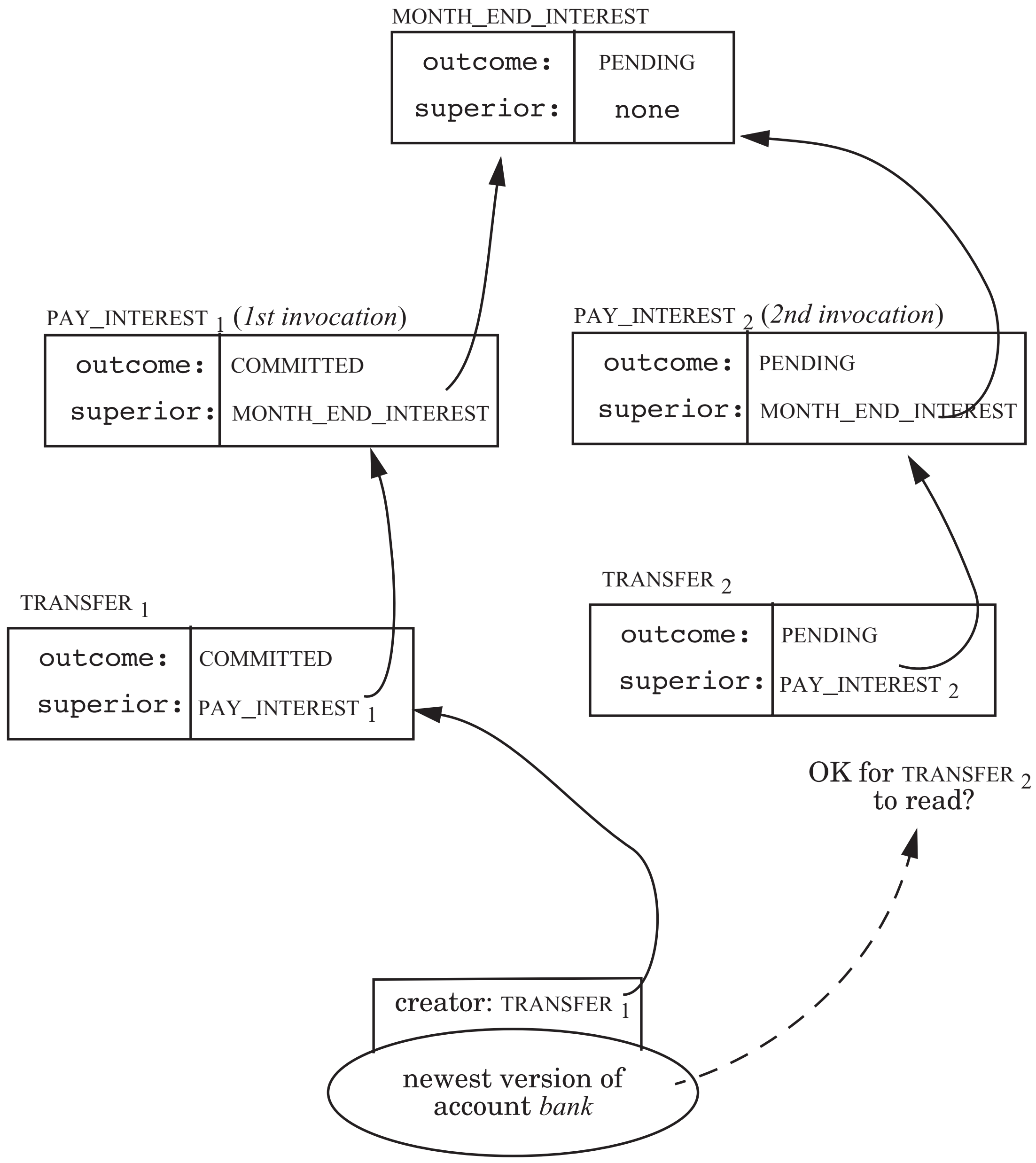


Figure 9–35: Transaction TRANSFER$_2$, nested in transaction PAY_INTEREST$_2$, which is nested in transaction MONTH_END_INTEREST, wants to read the current value of account *bank*. But *bank* was last written by transaction TRANSFER$_1$, which is nested in COMMITTED transaction PAY_INTEREST$_1$, which is nested in still-PENDING transaction MONTH_END_INTEREST. Thus this version of *bank* is actually PENDING, rather than COMMITTED as one might conclude by looking only at the outcome of TRANSFER$_1$. However, TRANSFER$_1$ and TRANSFER$_2$ share a common ancestor (namely, MONTH_END_INTEREST), and the chain of transactions leading from bank to that common ancestor is completely committed, so the read of *bank* can—and to avoid a deadlock, must—be allowed.

CREATE_NEW_VERSION as part of its plan to update the value of account “bank”, thereby establishing a serial order of the invocations. When later invocations of PAY_INTEREST call READ_CURRENT_VALUE to read the value of account “bank”, they will be forced to wait until all earlier invocations of PAY_INTEREST decide whether to commit or abort.

2. *Two-phase commit*

Since a higher-layer transaction can comprise several lower-layer transactions, we can describe the commitment of a hierarchical transaction as involving two distinct phases. In the first phase, known variously as the *preparation* or *voting* phase, the higher-layer transaction invokes some number of distinct lower-layer transactions, each of which either aborts or, by committing, becomes tentatively committed. The top-layer transaction evaluates the situation to establish that all (or enough) of the lower-layer transactions are tentatively committed that it can declare the higher-layer transaction a success.

Based on that evaluation, it either COMMITs or ABORTs the higher-layer transaction. Assuming it decides to commit, it enters the second, *commitment* phase, which in the simplest case consists of simply changing its own state from PENDING to COMMITTED or ABORTED. If it is the highest-layer transaction, at that instant all of the lower-layer tentatively committed transactions also become either COMMITTED or ABORTED. If it is itself nested in a still higher-layer transaction, it becomes tentatively committed and its component transactions continue in the tentatively committed state also. We are implementing here a coordination protocol known as *two-phase commit*. When we implement multiple-site atomicity in the next section, the distinction between the two phases will take on additional clarity.

If the system uses version histories for atomicity, the hierarchy of figure 9–35 can be directly implemented by linking outcome records. If the system uses logs, a separate table of pending transactions can contain the hierarchy, and inquiries about the state of a transaction would involve examining this table.

The concept of nesting transactions hierarchically is useful in its own right, but our particular interest in nesting is that it is the first of two building blocks for multiple-site transactions. To develop the second building block, we next explore what makes multiple-site transactions different from single-site transactions.

3. *Multiple-site atomicity: distributed two-phase commit*

If a transaction requires executing component transactions at several sites that are separated by a best-effort network, obtaining atomicity is more difficult, because any of the messages used to coordinate the transactions of the various sites can be lost, delayed, or duplicated. In chapter 4 we learned of a method, known as Remote Procedure Call (RPC) for performing an action at another site. In chapter 7 we learned how to design protocols such as RPC with a persistent sender to assure at-least-once execution and duplicate suppression to assure at-most-once execution. Unfortunately, neither of these two assurances is exactly what is needed to assure atomicity of a multiple-site transaction. However, by properly combining a two-phase commit protocol with persistent senders, duplicate suppression, and single-site transactions, we can create a correct multiple-site transaction. We assume that each site, on its own, is capable of implementing local transactions, using techniques such as version histories or logs and locks for all-or-nothing atomicity and isolation atomicity. Correctness of the multiple-site atomicity protocol will be achieved if all the sites commit or if all the sites abort; we will have failed if some sites commit their part of a multiple-site transaction while others abort their part of that same transaction.

Suppose the multiple-site transaction consists of a coordinator Alice requesting component transactions X, Y, and Z of worker sites Bob, Charles, and Dawn, respectively. The simple expedient of issuing three remote procedure calls certainly does not produce a transaction for Alice, because Bob may do X while Charles may report that he cannot do Y. Conceptually, the coordinator would like to send three messages, to the three workers, like this one to Bob:

From: Alice;
To: Bob;
Re: my transaction 91;

if (Charles does Y **and** Dawn does Z) **then do** X, please.

and let the three workers handle the details. We need some clue how Bob could accomplish this strange request.

The clue comes from recognizing that the coordinator has created a higher-layer transaction and each of the workers is to perform a transaction that is nested in the higher-layer transaction. Thus, what we need is a distributed version of the two-phase commit protocol. The complication is that the coordinator and workers cannot reliably communicate. The problem thus reduces to constructing a reliable distributed version of the two-phase commit protocol. We can do that by applying persistent senders and duplicate suppression.

Phase one of the protocol starts with coordinator Alice creating a top-layer outcome record for the overall transaction. Then Alice begins persistently sending to Bob an RPC-like message:

From: Alice;
To: Bob;
Re: my transaction 271;

Please do X as part of my transaction.

Similar messages go from Alice to Charles and Dawn, also referring to transaction 271, and requesting that they do Y and Z, respectively. As with an ordinary remote procedure call, if Alice doesn't receive a response from one or more of the workers in a reasonable time she resends the message to the non-responding workers as many times as necessary to elicit a response.

A worker site, upon receiving a request of this form, checks for duplicates and then creates a transaction of its own, but it makes the transaction a *nested* one, with its superior being Alice's original transaction. It then goes about doing the pre-commit part of the requested action, reporting back to Alice that this much has gone well:

> From: Bob;
> To: Alice;
> Re: your transaction 271;
>
> My part X is ready to commit.

Alice, upon collecting a complete set of such responses then moves to the two-phase commit part of the transaction, by sending messages to each of Bob, Charles, and Dawn saying, e.g.:

Two-phase-commit message #1:

> From: Alice;
> To: Bob;
> Re: my transaction 271;
>
> PREPARE to commit X.

Bob, upon receiving this message, commits—but only tentatively—or aborts. Having created durable tentative versions (or logged to journal storage its planned updates) and having recorded an outcome record saying that it is PREPARED either to commit or abort, Bob then persistently sends a response to Alice reporting his state:

Two-phase-commit message #2:

> From: Bob;
> To: Alice;
> Re: your transaction 271;
>
> I am PREPARED to commit my part. Have you decided to commit yet? Regards.

or alternatively, a message reporting it has aborted. If Bob receives a duplicate request from Alice, his persistent sender sends back a duplicate of the PREPARED or ABORTED response.

At this point Bob, being in the PREPARED state, is out on a limb. Just as in a local hierarchical nesting, Bob must be able either to run to the end or to abort, to maintain that state of preparation indefinitely, and wait for someone else (Alice) to say which. In addition, the coordinator may independently crash or lose communication contact, increasing Bob's uncertainty. If the coordinator goes down, all of the workers must wait until it recovers; in this protocol, the coordinator is a single point of failure.

As coordinator, Alice collects the response messages from her several workers (perhaps re-requesting PREPARED responses several times from some worker sites). If all workers send PREPARED messages, phase one of the two-phase commit is complete. If any worker responds with an abort message, or doesn't respond at all, Alice has the usual choice of aborting the entire transaction or perhaps trying a different worker site to carry out that component transaction. Phase two begins when Alice commits the entire transaction by marking her own outcome record COMMITTED.

Once the higher-layer outcome record is marked as COMMITTED or ABORTED, Alice sends a completion message back to each of Bob, Charles, and Dawn:

Two-phase-commit message #3:

> From: Alice;
> To: Bob;
> Re: my transaction 271;
>
> My transaction committed. Thanks for your help.

Each worker site, upon receiving such a message, changes its state from PREPARED to COMMITTED, performs any needed post-commit actions, and exits. Meanwhile, Alice can go about other business, with one important requirement for the future: she must remember, reliably and for an indefinite time, the outcome of this transaction. The reason is that one or more of her completion messages may have been lost. Any worker sites that are in the PREPARED state are awaiting the completion message to tell them which way to go. If a completion message does not arrive in a reasonable period of time, the persistent sender at the worker site will resend its PREPARED message. Whenever Alice receives a duplicate PREPARED message, she simply sends back the current state of the outcome record for the named transaction.

If a worker site that uses logs and locks crashes, the recovery procedure at that site has to take three extra steps. First, it must classify any PREPARED transaction as a tentative winner that it should restore to the PREPARED state. Second, if the worker is using locks for isolation, the recovery procedure must reacquire any locks the PREPARED transaction was holding at the time of the failure. Finally, the recovery procedure must restart the persistent sender, to learn the current status of the higher-layer transaction. If the worker site uses version histories, only the last step, restarting the persistent sender, is required.

Since the workers act as persistent senders of their PREPARED messages, Alice can be confident that every worker will eventually learn that her transaction committed. But since the persistent senders of the workers are independent, Alice has no way of assuring that they will act simultaneously. Instead, Alice is assured only of eventual completion of her transaction. This distinction between simultaneous action and eventual action is critically important, as will soon be seen.

If all goes well, two-phase commit of *N* worker sites will be accomplished in *3N* messages, as shown in figure 9–36: for each worker site a PREPARE message, a PREPARED message in response, and a COMMIT message. This *3N* message protocol is complete and sufficient, although there are several variations one can propose.

An example of a simplifying variation is that the initial RPC request and response could also carry the PREPARE and PREPARED messages, respectively. However, once a worker sends a PREPARED message, it loses the ability to unilaterally abort, and it must remain on the knife edge awaiting instructions from the coordinator. To minimize this wait, it is usually preferable to delay the PREPARE/PREPARED message pair until the coordinator knows that the other workers seem to be in a position to do their parts.

Some versions of the distributed two-phase commit protocol have a fourth acknowledgement message from the worker sites to the coordinator. The intent is to collect a

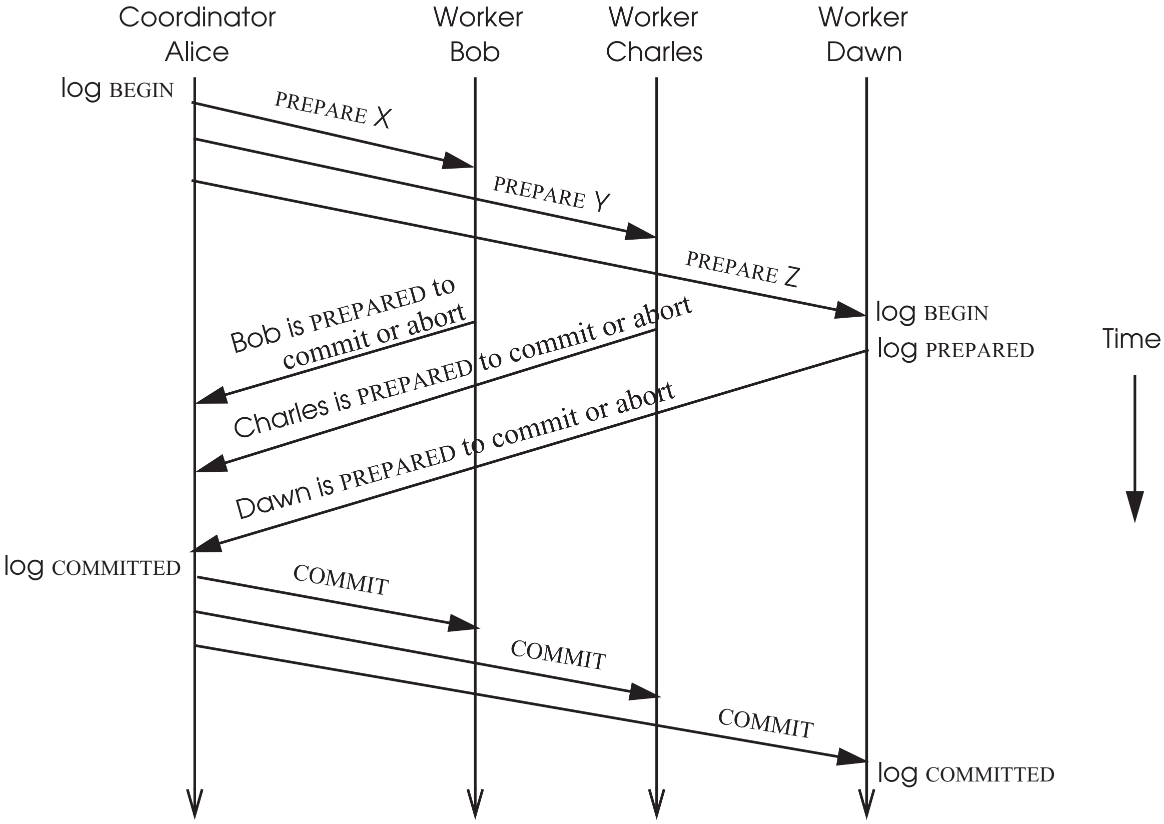


Figure 9–36: Timing diagram for distributed two-phase commit, using *3N* messages. (The initial RPC request and response messages are not shown.) Each of the four participants maintains its own version history or recovery log. The diagram shows log entries made by the coordinator and by one of the workers.

complete set of acknowledgement messages—the coordinator persistently sends completion messages until every site acknowledges. Once all acknowledgements are in, the coordinator can then safely discard its outcome record, since every worker site is known to have gotten the word.

A system that is concerned both about outcome record storage space and the cost of extra messages can use a further refinement, called *presumed commit*. Since one would expect that most transactions commit, we can use a slightly odd but very space-efficient representation for the value COMMITTED of an outcome record: non-existence. The coordinator answers any inquiry about a non-existent outcome record by sending a COMMITTED response. If the coordinator uses this representation, it commits by destroying the outcome record, so a fourth acknowledgement message from every worker is unnecessary. In return for this apparent magic reduction in both message count and space, we notice that outcome records for aborted transactions can not easily be discarded, because if an inquiry arrives after discarding, the inquiry will receive the response COMMITTED. The coordinator can, however, persistently ask for acknowledgement of aborted transactions, and discard the outcome record after all these acknowledgements are in. This protocol that leads to discarding an outcome record is identical to the protocol described in chapter 7 to close a stream and discard the record of that stream.

Distributed two-phase commit does not solve all multiple-site atomicity problems. For example, if the coordinator site (in this case, Alice) is aboard a ship that sinks after sending the PREPARE message but before sending the COMMIT or ABORT message the worker sites are in left in the PREPARED state with no way to proceed. Even without that concern, Alice and her co-workers are standing uncomfortably close to a multiple-site atomicity problem that, at least in principle, can *not* be solved. The only thing that rescues them is our observation that the several workers will do their parts eventually, not necessarily simultaneously. If she had required simultaneous action, Alice would have been in trouble.

The unsolvable problem is known as the *dilemma of the two generals*.

4. The dilemma of the two generals

An important constraint on possible coordination protocols when communication is unreliable is captured in a vivid analogy, called the *dilemma of the two generals*.* Suppose that two small armies are encamped on two mountains outside a city. The city is well-enough defended that it can repulse and destroy either one of the two armies. Only if the two armies attack simultaneously can they take the city. Thus the two generals who command the armies desire to coordinate their attack.

The only method of communication between the two generals is to send runners from one camp to the other. But the defenders of the city have sentries posted in the valley separating the two mountains, so there is a chance that the runner, trying to cross the valley, will instead fall into enemy hands, and be unable to deliver the message.

Suppose that the first general sends this message:

> From: Julius Caesar;
> To: Titus Labienus;
> Date: 11 January;
>
> I propose to cross the Rubicon and attack at dawn tomorrow. OK?

expecting that the second general will respond either with:

> From: Titus Labienus;
> To: Julius Caesar;
> Date: 11 January;
>
> Yes, dawn on the 12th.

* The origin of this analogy has been lost, but it was apparently first described in print in 1977 by Jim N. Gray in his "Notes on Database Operating Systems", reprinted in *Operating Systems, Lecture Notes in Computer Science 60*, Springer Verlag, 1978. At about the same time, Danny Cohen described another analogy he called the dating protocol, which is congruent with the dilemma of the two generals.

or, possibly:

> From: Titus Labienus;
> To: Julius Caesar;
> Date: 11 January;
>
> No. I am awaiting reinforcements from Gaul.

Suppose further that the first message does not make it through. In that case, the second general does not march because no request to do so arrives. In addition, the first general does not march because no response returns, and all is well (except for the lost runner).

Now, instead suppose the runner delivers the first message successfully and second general sends the reply “Yes,” but that the reply is lost. The first general cannot distinguish this case from the earlier case, so that army will not march. The second general has agreed to march, but knowing that the first general won’t march unless the “Yes” confirmation arrives, the second general will not march without being certain that the first general received the confirmation. This hesitation on the part of the second general suggests that the first general should send back an acknowledgement of receipt of the confirmation:

> From: Julius Caesar;
> To: Titus Labienus;
> Date: 11 January;
>
> The die is cast.

Unfortunately, that doesn’t help, since the runner carrying this acknowledgement may be lost and the second general, not receiving the acknowledgement, will still not march. Thus the dilemma.

We can now leap directly to a conclusion: there is no protocol with a bounded number of messages that can convince both generals that it is safe to march. If there were such a protocol, the *last* message in any particular run of that protocol must be unnecessary to safe coordination, because it might be lost, undetectably. Since the last message must be unnecessary, one could delete that message to produce another, shorter sequence of messages that must guarantee safe coordination. We can reapply the same reasoning repeatedly to the shorter message sequence to produce still shorter ones, and we conclude that if such a safe protocol exists it either generates message sequences of zero length or else of unbounded length. A zero-length protocol can’t communicate anything, and an unbounded protocol is of no use to the generals, who must choose a particular time to march.

A practical general, presented with this dilemma by a mathematician in the field, would reassign the mathematician to a new job as a runner, and send a scout to check out the valley and report the probability that a successful transit can be accomplished within a specified time. Knowing that probability, the general would then send several (hopefully independent) runners, each carrying a copy of the message, choosing a number of runners large enough that the probability is negligible that all of them fail to deliver the message before the appointed time. (The loss of all the runners would be what chapter 8 called an intolerable error.) Similarly, the second general sends many runners each carrying a copy of either the “Yes” or the “No” acknowledgement. This procedure provides a practical solution of the problem, so the dilemma is of no real consequence. Nevertheless, it is interesting to discover a problem that cannot, in principle, be solved with complete certainty.

We can state the theoretical conclusion more generally and succinctly: if messages may be lost, no bounded protocol can guarantee with complete certainty that both generals know that they will both march at the same time. The best that they can do is accept some non-zero probability of failure equal to the probability of non-delivery of their last message.

It is interesting to analyze just why we can't we use a distributed two-phase commit protocol to resolve the dilemma of the two generals. As suggested at the outset, it has to do with a subtle difference in *when* things may, or must, happen. The two generals require, in order to vanquish the defenses of the city, that they march at the *same* time. The persistent senders of the distributed two-phase commit protocol assure that if the coordinator decides to commit, all of the workers will eventually also commit, but there is no assurance that they will do so at the same time. If one of the communication links goes down for a day, when it comes back up the worker at the other end of that link will then receive the notice to commit, but this action may occur a day later than the actions of its colleagues. Thus the problem solved by distributed two-phase commit is slightly relaxed when compared with the dilemma of the two generals. That relaxation doesn't help the two generals, but the relaxation turns out to be just enough to allow us to devise a protocol that assures correctness.

By a similar line of reasoning, there is no way to assure with complete certainty that actions will be taken simultaneously at two sites that communicate only via a best-effort network. Distributed two-phase commit can thus safely open a cash drawer of an ATM in Tokyo, with confidence that a computer in Munich will eventually update the balance of that account. But if, for some reason, it is necessary to open two cash drawers at different sites at the same time, the only solution is either the probabilistic approach or to somehow replace the best-effort network with a reliable one. The requirement for reliable communication is why real estate transactions and weddings (both of which are examples of two-phase commit protocols) usually occur with all of the parties in one room.

Appendix 9–A. Case studies of machine language atomicity

1. *Complex instruction sets: The General Electric 600 line*

In the early days of mainframe computers, most manufacturers reveled in providing elaborate instruction sets, without paying much attention to questions of atomicity. The General Electric 600 line, which later evolved to be the Honeywell Information System, Inc., 68 series computer architecture, had a feature called "indirect and tally." One could specify this feature by setting to ON a one-bit flag (the "tally" flag) stored in an unused high-order bit of any indirect address. The instruction

Load register A from *Y* indirect.

was interpreted to mean that the low-order bits of the cell with address *Y* contain another address, called an indirect address, and that indirect address should be used to retrieve the operand to be loaded into register A. In addition, if the tally flag in cell *Y* is ON, the processor is to increment the indirect address in *Y* by one and store the result back in *Y*. The idea is that the next time *Y* is used as an indirect address it will point to a different operand—the one in the next sequential address in memory. Thus the indirect and tally feature could be used to sweep through a table. The feature seemed useful to the designers, but it was actually only occasionally, because most applications were written in higher-level languages and compiler writers found it hard to exploit. On the other hand the feature gave no end of trouble when virtual memory was retrofitted to the product line.

Suppose that virtual memory is in use, and that the indirect word is located in a page that is in primary memory, but the actual operand is in another page that has been removed to secondary memory. When the above instruction is executed, the processor will retrieve the indirect address in *Y*, increment it, and store the new value back in *Y*. Then it will attempt to retrieve the actual operand, at which time it discovers that it is not in primary memory, so it signals a missing-page exception. Since it has already modified the contents of *Y* (and by now *Y* may have been read by another processor or even removed from memory by the missing-page exception handler running on another processor), it is not feasible to back out and act as if this instruction had never executed. The designer of the exception handler would like to be able to give the processor to another thread by calling a function such as AWAIT while waiting for the missing page to arrive. Indeed, processor reassignment may be the only way to assign a processor to retrieve the missing page. However, to reassign the processor it is necessary to save its current execution state. Unfortunately, its execution state is "half-way through the instruction last addressed by the program counter." Saving this state and later restarting the processor in this state is challenging. The indirect and tally feature was just one of several sources of atomicity problems that cropped up when virtual memory was added to this processor.

The virtual memory designers desperately wanted to be able to run other threads on the interrupted processor. To solve this problem, they extended the definition of the current program state to contain not just the next-instruction counter and the program-visible registers, but also the complete internal state description of the processor—a 216-bit snapshot in the middle of the instruction. By later restoring the processor state to contain the previously saved values of the next-instruction counter, the program-visible registers, and the 216-bit internal state snapshot, the processor could exactly continue from the point at which the missing-page alert occurred. This technique worked but it had two awkward side effects: 1) when a program (or programmer) inquires about the current state of an interrupted processor, the state description includes things not in the programmer's interface; and 2) the system must be very careful when restarting an interrupted program to make certain that the stored micro-state description is a valid one. If someone has altered the state description the processor could try to continue from a state it could never have gotten into by itself, which could lead to unplanned behavior, including failures of its memory protection features.

2. *More elaborate instruction sets: The IBM System/370*

When IBM developed the System/370 by adding virtual memory to its System/360 architecture, certain System/360 multi-operand character-editing instructions caused atomicity problems. For example, the TRANSLATE instruction contains three arguments, two of which are addresses in memory (call them *string* and *table*) and the third of which, *length*, is an 8-bit count that the instruction interprets as the length of *string*. TRANSLATE takes one byte at a time from *string*, uses that byte as an offset in *table*, retrieves the byte at the offset, and replaces the byte in *string* with the byte it found in *table*. The designers had in mind that TRANSLATE could be used to convert a character string from one character set to another.

The problem with adding virtual memory is that both *string* and *table* may be as long as 65,536 bytes, so either or both of those operands may cross not just one, but several page boundaries. Suppose just the first page of *string* is in physical memory. The TRANSLATE instruction works its way through the bytes at the beginning of string. When it comes to the end of that first page, it encounters a missing-page exception. At this point, the instruction cannot run to completion because data it requires is missing. It also cannot back out and act as if it never started, because it has modified data in memory by overwriting it. After the virtual memory manager retrieves the missing page, the problem is how to restart the half-completed instruction. If it restarts from the beginning, it will try to convert the already-converted characters, which would be a mistake. For correct operation, the instruction needs to continue from where it left off.

Rather than tampering with the program state definition, the IBM processor designers chose a *dry run* strategy in which the TRANSLATE instruction is executed using a hidden copy of the program-visible registers and making no changes in memory. If one of the operands causes a missing-page exception, the processor can act as if it never tried the instruction, since there is no program-visible evidence that it did. The stored program state shows only that the TRANSLATE instruction is about to be executed. After the processor retrieves the missing page, it restarts the interrupted thread by trying the TRANSLATE instruction from the beginning again, another dry run. If there are several missing pages, several dry runs may occur, each getting one more page into primary memory. When a dry run finally succeeds in completing, the processor runs the instruction once more, this time for real, using the program-visible registers and allowing memory to be updated. Since the System/370 (at the time this modification was made) was a single-processor architecture, there was no possibility that another processor might snatch a page away after the dry run but before the real execution of the instruction. This solution had the side effect of making life more difficult for a later designer with the task of adding multiple processors.

3. The Apollo desktop computer and the Motorola M68000 microprocessor

When Apollo Computer designed a desktop computer using the Motorola 68000 microprocessor, the designers, who wanted to add a virtual memory feature, discovered that the microprocessor instruction set interface was not atomic. Worse, because it was constructed entirely on a single chip it could not be modified to do a dry run (as in the IBM 370) or to make it store the internal microprogram state (as in the General Electric 600 line). So the Apollo designers used a different strategy: they installed not one, but two Motorola 68000 processors. When the first one encounters a missing-page exception, it simply stops in its tracks, and waits for the operand to appear. The second Motorola 68000 (whose program is carefully planned to reside entirely in primary memory) fetches the missing page and then restarts the first processor.

Other designers working with the Motorola 68000 used a different, somewhat risky trick: modify all compilers and assemblers to generate only instructions that happen to be atomic. Motorola later produced a version of the 68000 in which all internal state registers of the microprocessor could be saved, the same method used in adding virtual memory to the General Electric 600 line.

Appendix 9–B. A better model of disk failure

Section B of this chapter developed a failure analysis model for a calendar management program in which a system crash may corrupt at most one disk sector—the one, if any, that was being written at the instant of the crash. That section also developed a masking strategy for that problem, creating all-or-nothing disk storage. To keep that development simple, the strategy ignored decay events. This appendix revisits that model, considering how to also mask decay events. The result will be all-or-nothing durable storage, meaning that it is both all-or-nothing in the event of a system crash and durable in the face of decay events.

1. Algorithms to obtain storage that is both all-or-nothing and durable

In chapter 8 we learned that to obtain durable storage we should write two or more replicas of each disk sector. In the current chapter we learned that to recover from a system crash while writing a disk sector we should never overwrite the previous version of that sector, we should write a new version in a different place. To obtain storage that is both durable and all-or-nothing we combine these two observations: make more than one replica, and don't overwrite the previous version. One easy way to do that would be to simply build the all-or-nothing storage layer of the current chapter on top of the durable storage layer of chapter 8. That method would certainly work but it is a bit heavy-handed: with a replication count of just two, it would lead to allocating six disk sectors for each sector of real data. This is a case in which modularity has an excessive cost.

Recall that the parameter that chapter 8 used to determine frequency of checking the integrity of disk storage was the expected time to decay, T_d. Suppose for the moment that the durability requirement can be achieved by maintaining only two copies. In that case, T_d must be much greater than the time required to write two copies of a sector on two disks. Put another way, a large T_d means that the short-term chance of a decay event is small enough that the designer may be able to safely neglect it. We can take advantage of this observation to devise a very slightly risky but far more economical method of implementing storage that is both durable and all-or-nothing with just two replicas. The basic idea is that if we are confident that we have two good replicas of some piece of data for durability, it is safe (for all-or-nothing atomicity) to overwrite one of the two replicas; the second replica can be used as a backup to assure all-or-nothing atomicity if the system should happen to crash while writing the first one. Once we are confident that the first replica has been correctly written with new data, we can safely overwrite the second one, to regain long-term durability. If the time to complete the two writes is short compared with T_d, the probability that a decay event interferes with this algorithm will be negligible. Figure 9–37 shows the algorithm and the two replicas of the data, here named *D0* and *D1*.

An interesting point is that ALL_OR_NOTHING_DURABLE_GET does not bother to check the status returned upon reading *D1*—it just passes the status value along to its caller. The reason is that in the absence of decay CAREFUL_GET has *no* expected errors when reading data

```
procedure ALL_OR_NOTHING_DURABLE_GET (reference data, atomic_sector)
{
    ds ← CAREFUL_GET (data, atomic_sector.D0);
    if ds = BAD then
        ds ← CAREFUL_GET (data, atomic_sector.D1);
    return ds;
}

procedure ALL_OR_NOTHING_DURABLE_PUT (new_data, atomic_sector)
{
    SALVAGE(atomic_sector);
    ds ← CAREFUL_PUT (new_data, atomic_sector.D0);
    ds ← CAREFUL_PUT (new_data, atomic_sector.D1);
    return ds;
}

procedure SALVAGE(atomic_sector)
{                                                   //Run this program every T_d seconds.
    ds0 ← CAREFUL_GET (data0, atomic_sector.D0);
    ds1 ← CAREFUL_GET (data1, atomic_sector.D1);
    if ds0 = BAD then
        CAREFUL_PUT (data1, atomic_sector.D0);
    else if ds1 = BAD then
        CAREFUL_PUT (data0, atomic_sector.D1);
    if data0 ≠ data1 then
        CAREFUL_PUT (data0, atomic_sector.D1);
}
```

D0:
DATA
D1:
DATA

Figure 9–37: Data arrangement and algorithms to implement all-or-nothing durable storage on top of the careful storage layer of figure 8–12 on page 8–55.

that CAREFUL_PUT was allowed to finish writing. Thus the returned status would be BAD only in two cases:

1. CAREFUL_PUT of *D1* was interrupted in mid-operation, or

2. *D1* was subject to an unexpected decay.

The algorithm guarantees that the first case cannot happen. ALL_OR_NOTHING_DURABLE_PUT doesn't begin CAREFUL_PUT on data *D1* until after the completion of its CAREFUL_PUT on data *D0*. At most one of the two copies could be BAD because of a system crash during CAREFUL_PUT. Thus if the first copy (*D0*) is BAD, then we expect that the second one (*D1*) is OK.

The risk of the second case is real, but we have assumed its probability to be small: it arises only if there is a random decay of *D1* in a time much shorter than T_d. In reading *D1* we have an opportunity to *detect* that error through the status value, but we have no way to recover when both data copies are damaged, so this detectable error must be classified as untolerated. All we can do is pass a status report along to the application so that it knows that there was an untolerated error.

There is one currently unnecessary step hidden in the SALVAGE program: if *D0* is BAD, nothing is gained by copying *D1* onto *D0*, since ALL_OR_NOTHING_DURABLE_PUT, which called SALVAGE, will immediately overwrite *D0* with new data. The step is included because it allows SALVAGE to be used in a refinement of the algorithm.

In the absence of decay events, this algorithm would be just as good as the all-or-nothing procedures of figures 9–6 and 9–7, and it would perform somewhat better, since it involves only two copies. Assuming that errors are rare enough that recovery operations do not dominate performance, the usual cost of ALL_OR_NOTHING_DURABLE_GET is just one disk read, compared with three in the ALL_OR_NOTHING_GET algorithm. The cost of ALL_OR_NOTHING_DURABLE_PUT is two disk reads (in SALVAGE) and two disk writes, compared with three disk reads and three disk writes for the ALL_OR_NOTHING_PUT algorithm.

That analysis is based on a decay-free system. To deal with decay events, thus making the scheme both all-or-nothing *and* durable, the designer adopts two ideas from the discussion of durability in chapter 8, the second of which eats up some of the better performance:

1. Place the two copies, *D0* and *D1*, in independent decay sets (for example write them on two different disk drives, preferably from different vendors).

2. Have a clerk run the SALVAGE program on every atomic sector at least once every T_d seconds.

The clerk running the SALVAGE program performs $2N$ disk reads every T_d seconds to maintain N durable sectors. This extra expense is the price of durability against disk decay. The performance cost of the clerk depends on the choice of T_d, the value of N, and the priority of the clerk. Since the expected operational lifetime of a hard disk is usually several years, setting T_d to a few weeks should make the chance of untolerated failure from decay negligible, especially if there is also an operating practice to routinely replace disks well before they reach their expected operational lifetime. A modern hard disk with a capacity of one terabyte would have about $N = 10^9$ kilobyte-sized sectors. If it takes 10 milliseconds to read a sector, it would take about 2×10^7 seconds, or two days, for a clerk to read all of the contents of two one-terabyte hard disks. If the work of the clerk is scheduled to occur at night, or uses a priority system that runs the clerk when the system is otherwise not being used heavily, that reading can spread out over a few weeks and the performance impact can be minor.

A few paragraphs back mentioned that there is the potential for a refinement: If we also run the SALVAGE program on every atomic sector immediately following every system crash, then it should not be necessary to do it at the beginning of every ALL_OR_NOTHING_DURABLE_PUT. That variation, which is more economical if crashes are infrequent and disks are not too large, is due to Butler Lampson and Howard Sturgis*. It raises one minor concern: it depends on the rarity of coincidence of two failures: the spontaneous decay of one data replica at about the same time that CAREFUL_PUT crashes in the middle of rewriting the other replica of that same sector. If we are convinced that such a coincidence is rare, we can declare it to be an untolerated error, and we have a self-consistent and more economical algorithm. With this scheme the cost of ALL_OR_NOTHING_DURABLE_PUT reduces to just two disk writes.

[*] Butler W. Lampson and Howard Sturgis. Crash recovery in a distributed data storage system. [Suggestions for Further Reading 3.2.8]