Principles of Computer System Design

An Introduction

Chapter 9
Atomicity: All-or-Nothing and Before-or-After

Jerome H. Saltzer

M. Frans Kaashoek

Massachusetts Institute of Technology

Version 5.0

Suggestions, Comments, Corrections, and Requests to waive license restrictions: Please send correspondence by electronic mail to:

Saltzer@mit.edu

and

kaashoek@mit.edu

CHAPTER

Atomicity: All-or-Nothing and Before-or-After

9

CHAPTER CONTENTS

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 *before-or-after atomicity*, a design strategy for coordinating concurrent activities. Chapter 8[on-line] 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 ensure that actions have the before-or-after property. This chapter explores ways to systematically synthesize a design that provides both the all-or-nothing property needed for failure masking and the before-or-after property needed for coordination.

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 credit card 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 activity initiated by their own click of the "purchase" button occur either completely before or completely after any other, concurrent click of a "purchase" button.

The single conceptual framework of atomicity provides a powerful way of thinking about both all-or-nothing failure masking and before-or-after sequencing 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—thus an atomic action provides all-or-nothing atomicity. Similarly, when several atomic actions are going on concurrently, each atomic action will appear to take place either completely before or completely after every other atomic action—thus an atomic action provides before-or-after atomicity. Together, all-or-nothing atomicity and before-or-after atomicity provide a particularly strong form of modularity: they hide the fact that the atomic action is actually composed of multiple steps.

The result is a *sweeping simplification* 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 ensure correct coordination *a priori* 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.

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 *before-or-after* (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 before-or-after, 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 before-or-after atomicity.

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

All-or-nothing atomicity and before-or-after 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[on-line] then explores how a designer can use transactions to implement constraints and enhance durability.

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[on-line] then explores some applications of atomicity. Case studies at the end of both chapters provide real-world examples of atomicity as a tool for creating useful systems.

9.1 Atomicity

Atomicity is a property required in several different areas of computer system design. These areas include managing a database, developing a hardware architecture, specifying the interface to an operating system, and more generally in software engineering. The table below suggests some of the kinds of problems to which atomicity is applicable. In this chapter we will encounter examples of both kinds of atomicity in each of these different areas.

Area	All-or-nothing atomicity	Before-or-after atomicity
database management	updating more than one record	records shared between threads
hardware architecture	handling interrupts and exceptions	register renaming
operating systems	supervisor call interface	printer queue
software engineering	handling faults in layers	bounded buffer

9.1.1 All-or-Nothing Atomicity in a Database

As a first example, consider a database of bank accounts. We define a procedure named TRANSFER that debits one account and credits a second account, both of which are stored on disk, as follows:

```
1   procedure TRANSFER (debit_account, credit_account, amount)
2       GET (dbdata, debit_account)
3       dbdata ← dbdata - amount
4       PUT (dbdata, debit_account)
5       GET (crdata, credit_account)
6       crdata ← crdata + amount
7       PUT (crdata, credit_account)
```

where *debit_account* and *credit_account* identify the records for the accounts to be debited and credited, respectively.

Suppose that the system crashes while executing the PUT instruction on line 4. Even if we use the MORE_DURABLE_PUT described in Section 8.5.4, a system crash at just the wrong time may cause the data written to the disk to be scrambled, and the value of *debit_account* lost. We would prefer that either the data be completely written to the disk or nothing be written at all. That is, we want the PUT instruction to have the all-or-nothing atomicity property. Section 9.2.1 will describe a way to do that.

There is a further all-or-nothing atomicity requirement in the TRANSFER procedure. Suppose that the PUT on line 4 is successful but that while executing line 5 or line 6 the power fails, stopping the computer in its tracks. When power is restored, the computer restarts, but volatile memory, including the state of the thread that was running the TRANSFER procedure, has been lost. If someone now inquires about the balances in *debit_account* and in *credit_account* things will not add up properly because *debit_account* has a new value but *credit_account* has an old value. One might suggest postponing the first PUT to be just before the second one, but that just reduces the window of vulnerability, it does not eliminate it—the power could still fail in between the two PUTs. To eliminate the window, we must somehow arrange that the two PUT instructions, or perhaps even the entire TRANSFER procedure, be done as an all-or-nothing atomic action. In Section 9.2.3 we will devise a TRANSFER procedure that has the all-or-nothing property, and in Section 9.3 we will see some additional ways of providing the property.

9.1.2 All-or-Nothing Atomicity in the Interrupt Interface

A second application for all-or-nothing atomicity is in the processor instruction set interface as seen by a thread. 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.

If the event is a signal arriving from outside the interpreter, the interrupt thread may simply invoke a thread management primitive such as ADVANCE, as described in Section 5.6.4, to alert some other thread about the event. For example, an I/O operation that the other thread was waiting for may now have completed. The interrupt handler then returns control to the interrupted thread. This example requires before-or-after atomicity between the interrupt thread and the interrupted thread. If the interrupted thread was in the midst of a call to the thread manager, the invocation of ADVANCE by the interrupt thread should occur either before or after that call.

Another possibility is that the interpreter has detected that something is going wrong in the interrupted thread. In that case, 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 exception handler will need to examine 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 of the state: 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 exception handler runs. 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 that new address, and adds that data 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 is problematic because after the handler retrieves the missing page it cannot simply tell the processor to jump to the instruction that failed—that would increment the address register again, which is not what the program-

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.

mer expected. Jumping to the next instruction isn't right, either, because that would omit the addition step. 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. Modern processor designers are generally careful to avoid designing instructions that don't have the all-or-nothing property. As will be seen shortly, designers of higher-layer interpreters must be similarly careful.

Sections 9.1.3 and 9.1.4 explore the case in which the exception terminates the running thread, thus creating a fault. Section 9.1.5 examines the case in which the interrupted thread continues, oblivious (one hopes) to the interruption.

9.1.3 All-or-Nothing Atomicity in a Layered Application

A third example of all-or-nothing atomicity lies in the challenge presented by a fault in a running program: 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 with the all-or-nothing property. 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 9.19.1 (that figure may seem familiar—it is a copy of Figure 2.10). 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 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.

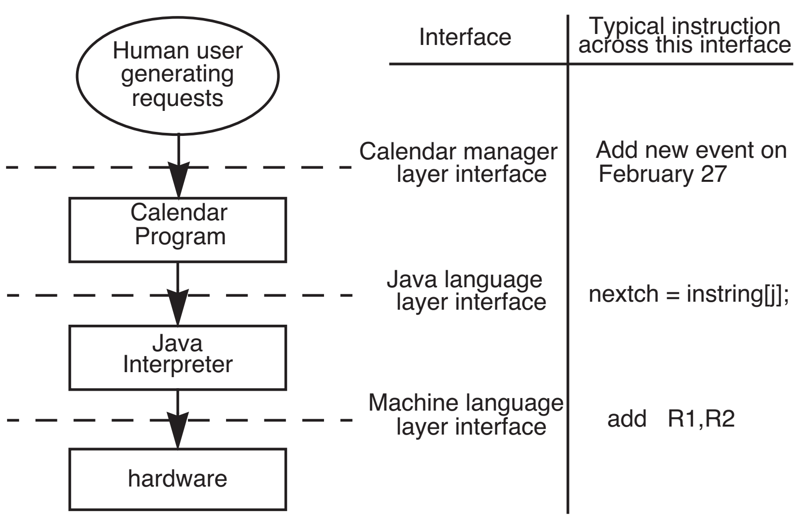


FIGURE 9.1

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.

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.

With the insight that in a layered application, we want a fault detected by a lower layer to be contained in a particular way we can now propose a more formal definition of all-or-nothing atomicity:

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.***

In a layered application, 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 (e.g. "there is no room in the calendar for another event") might be intercepted by some intermediate layer that knows of a way to mask it (e.g., by allocating more storage space). 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.

9.1.4 Some Actions With and Without the All-or-Nothing Property

Actions that lack the all-or-nothing property have frequently been 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. Section 9.8 describes some examples of machine architectures that had this problem and the techniques that were used to add virtual memory to them.

A second example is the supervisor call (SVC). Section 5.3.4 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 ensure 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 passes 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, causing AWAIT to return, the user's thread re-executes the original kernel call to READ, this time finding the typed

input. With this design, if a 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 ensures 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 exception 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.

9.1.5 Before-or-After Atomicity: 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 9.1.2 above 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 *before-or-after atomicity*. 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 actions. Sequence coordination thus is usually explicitly programmed, using either special language constructs or shared variables such as the eventcounts of Chapter 5.

In contrast, *before-or-after atomicity* is a more general constraint that several actions that concurrently operate on the same data should not interfere with one another. We define before-or-after atomicity as follows:

Before-or-after atomicity

Concurrent actions have the *before-or-after* property if their effect from the point of view of their invokers is the same as if the actions occurred either *completely before* or *completely after* one another.

In Chapter 5 we saw how before-or-after actions can be created with explicit locks and a thread manager that implements the procedures ACQUIRE and RELEASE. Chapter 5 showed some examples of before-or-after actions using locks, and emphasized that programming correct before-or-after actions, for example coordinating a bounded buffer with several producers or several consumers, can be a tricky proposition. To be confident of correctness, one needs to establish a compelling argument that every action that touches a shared variable follows the locking protocol.

One thing that makes before-or-after atomicity different from sequence coordination is that the programmer of an action that must have the before-or-after property does not necessarily know the identities of all the other actions that might touch the shared variable. This lack of knowledge can make it problematic to coordinate actions by explicit program steps. Instead, what the programmer needs is an automatic, implicit mechanism that ensures proper handling of every shared variable. This chapter will describe several such mechanisms. Put another way, correct coordination requires discipline in the way concurrent threads read and write shared data.

Applications for before-or-after atomicity in a computer system abound. In an operating system, several concurrent threads may decide to use a shared printer at about the same time. It would not be useful for printed lines of different threads to be interleaved in the printed output. Moreover, it doesn't really matter which thread gets to use the printer first; the primary consideration is that one use of the printer be complete before the next begins, so the requirement is to give each print job the before-or-after atomicity property.

For a more detailed example, let us return to the banking application and the TRANSFER procedure. This time the account balances are held in shared memory variables (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
```

Despite their unitary appearance, a program statement such as "*X* ← *X* + *Y*" is actually composite: it involves 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*.

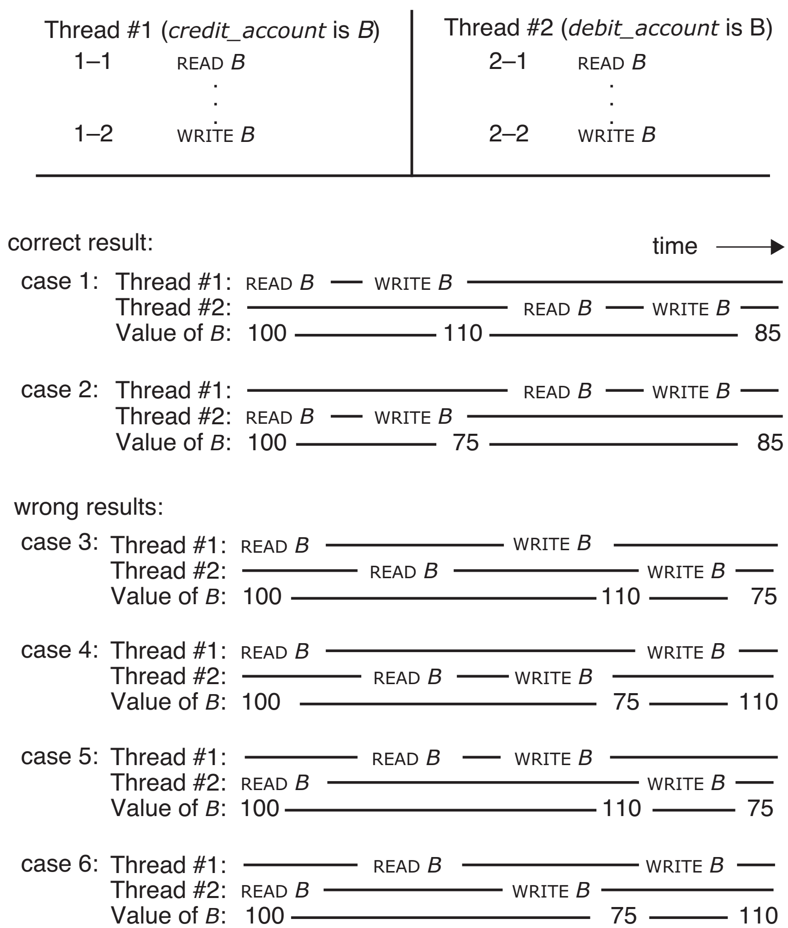


FIGURE 9.2

Six possible histories of variable *B* if two threads that share *B* do not coordinate their concurrent activities.

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.

9.1.6 Correctness and Serialization

The notion that the first two sequences of Figure 9.2 are correct and the other four are wrong is based on our understanding of the banking 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 mechanism that provides before-or-after atomicity 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:

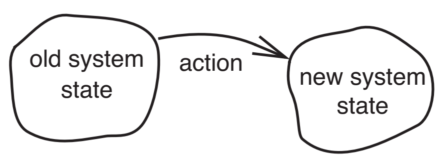


FIGURE 9.3

A single action takes a system from one state to another 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,

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.
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.

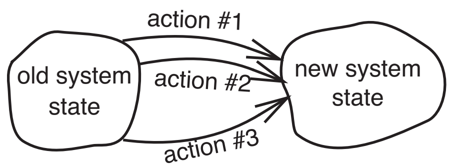


FIGURE 9.4

When several actions act concurrently, they together produce a new state. If the actions are before-or-after and the old state was correct, the new state will be correct.

So long as the only coordination requirement is before-or-after atomicity, any serialization will do.

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 correct, 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 correct because there exists a trajectory that leads to that state for which a correctness argument could have been applied to every step.

Since our definition of before-or-after atomicity is that each before-or-after action act as though it ran either completely before or completely after each other before-or-after action, before-or-after atomicity leads directly to this concept of correctness. Put another way, before-or-after atomicity has the effect of serializing the actions, so it follows that before-or-after atomicity guarantees correctness of coordination. A different way of

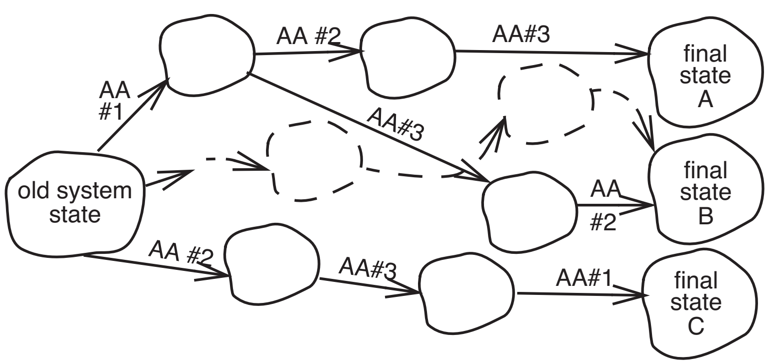


FIGURE 9.5

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*.

expressing this idea is to say that when concurrent actions have the before-or-after property, they are *serializable: there exists some serial order of those concurrent transactions that would, if followed, lead to the same ending state.*[*] 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 the example of Figure 9.2, there were only two concurrent actions and each of the concurrent actions had only two steps. As the number of concurrent actions and the number of steps in each action grows there will be a rapidly growing number of possible orders in which the individual steps can occur, but only some of those orders will ensure a correct result. Since the purpose of concurrency is to gain performance, one would like to have a way of choosing from the set of correct orders the one correct order that has the highest performance. As one might guess, making that choice can in general be quite difficult. In Sections 9.4 and 9.5 of this chapter we will encounter several programming disciplines that ensure choice from a subset of the possible orders, all members of which are guaranteed to be correct but, unfortunately, may not include the correct order that has the highest performance.

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 what might be called *external time consistency*: if there is any external evidence (such as a printed receipt) that before-or-after action T_1 ended before before-or-after 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 before-or-after atomicity requirements. Consider the following auditing procedure; its purpose is to verify that the sum of the balances of all accounts is zero (in double-entry bookkeeping, accounts belonging to the bank, such as the amount of cash in the vault, have negative balances):

```
procedure AUDIT()
    sum ← 0
    for each W ← in bank.accounts
        sum ← sum + W.balance
    if (sum ≠ 0) call for investigation
```

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

[*] The general question of whether or not a collection of existing transactions is serializable is an advanced topic that is addressed in database management. Problem set *36* explores one method of answering this question.

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 a before-or-after action.

There is yet another before-or-after 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 should occur either completely before or completely after any AUDIT action; put another way, TRANSFER should be a before-or-after action.

9.1.7 All-or-Nothing and Before-or-After Atomicity

We now have seen examples of two forms of atomicity: all-or-nothing and before-or-after. These two forms have a common underlying goal: to hide the internal structure of an 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 the internal structure of its implementation.

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 before-or-after 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 internal structure 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 has structure. 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 internal structure of data; client/server organization has the goal of hiding the internal structure of major subsystems. Similarly, atomicity has the goal of hiding the internal structure of an action. 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 hide this reality from the higher layer and from 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 internal structure of atomic actions: atomic actions can have benevolent side effects. A common example is an audit log, where 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 record 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.

9.2 All-or-Nothing Atomicity I: Concepts

Section 9.1 of this chapter defined the goals of all-or-nothing atomicity and before-or-after atomicity, and provided a conceptual framework that at least in principle allows a designer to decide whether or not some proposed algorithm correctly coordinates 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 before-or-after 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 creating before-or-after actions from arbitrary sequences of code by implementing a procedure named ACQUIRE that itself required before-or-after atomicity of two or three lines of code where it reads and then sets a lock value. It then implemented that before-or-after action with the help of a special hardware feature that directly makes a before-or-after action of the read and set sequence, and it also exhibited a software implementation (in Sidebar 5.2) that relies only on the hardware performing ordinary LOADS and STORES as before-or-after actions. 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. The reduction method, called the *version history*, is used only occasionally in practice, but once understood it becomes easy to see why the more widely used reduction methods that will be described in Section 9.3 work.

9.2.1 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[on-line], 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[on-line]. 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 Section 8.5.4.2: 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[on-line]—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 Section 8.5.4.5), 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 will reverse it in Section 9.7, 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 can classify the error as "detectable" if 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 situa-

```
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.

tion: 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 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 (in other words, all three copies are identical), 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 ensured 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,

```
procedure ALL_OR_NOTHING_PUT (data, all_or_nothing_sector)
    CHECK_AND_REPAIR (all_or_nothing_sector)
    ALMOST_ALL_OR_NOTHING_PUT (data, all_or_nothing_sector)

procedure CHECK_AND_REPAIR (all_or_nothing_sector) // Ensure copies match.
    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) and (data2 = data3) return     // State 1 or 7, no repair
    if (data1 = data2)
        CAREFUL_PUT (data1, all_or_nothing_sector.S3) return    // State 5 or 6.
    if (data2 = data3)
        CAREFUL_PUT (data2, all_or_nothing_sector.S1) return    // State 2 or 3.
    CAREFUL_PUT (data1, all_or_nothing_sector.S2)      // State 4, go to state 5
    CAREFUL_PUT (data1, all_or_nothing_sector.S3)      // State 5, go to state 7
```

FIGURE 9.7

Algorithms for ALL_OR_NOTHING_PUT and CHECK_AND_REPAIR.

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.

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 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 8 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 9), the data must be in state 5 or state 6, so CHECK_AND_REPAIR forces sector *S3* to match and returns (line 10). If the copies in sectors *S2* and *S3* are identical the data must be in state 2 or state 3 (line 11), so CHECK_AND_REPAIR forces sector *S1* to match and returns (line 12). The only remaining possibility is that the data is in state 4, in which case sector *S2* is surely bad, but sector *S1* contains a new value and sector *S3* contains an old 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 before-or-after 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 8.5. 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[on-line] 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 Section 9.7, 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.

9.2.2 Systematic 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, to ensure 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.

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 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*.

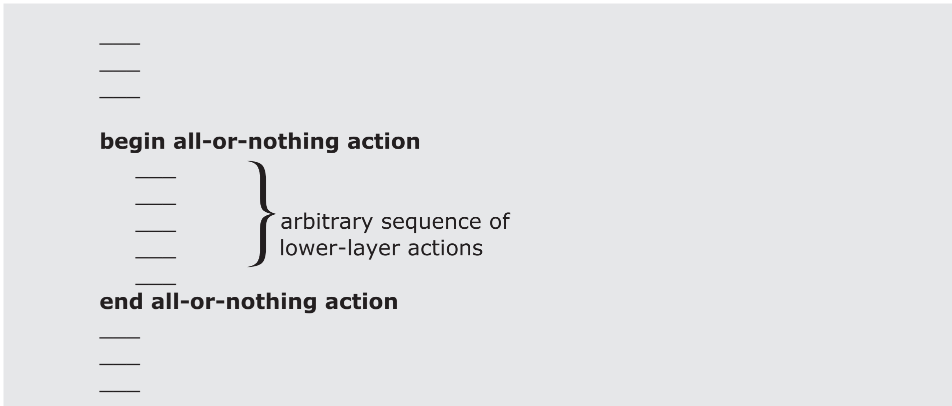


FIGURE 9.8

Imaginary semantics for painless programming of all-or-nothing actions.

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

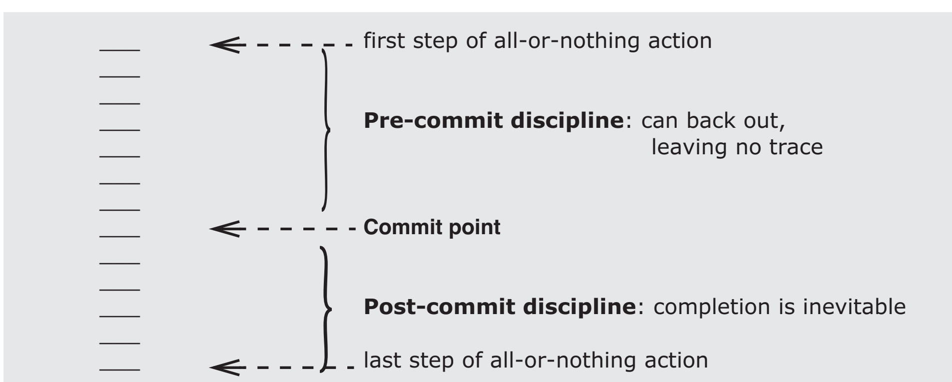


FIGURE 9.9

The commit point of an all-or-nothing action.

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 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 ensure 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 ensure 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 ensures 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.

Sidebar 9.3: Cascaded aborts *(Temporary)* <u>*sweeping simplification*</u>. In this initial discussioin 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[online] introduces a simple version of cascaded abort.

- Commit point: 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 in Section 2.5.8.
- Post-commit: 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 will 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.

9.2.3 Systematic All-or-Nothing Atomicity: Version Histories

This section 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 9.3 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 before-or-after atomicity will also emerge, we leave a systematic treatment of that topic for discussion in Sections 9.4 and 9.5 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[on-line] 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 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, 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

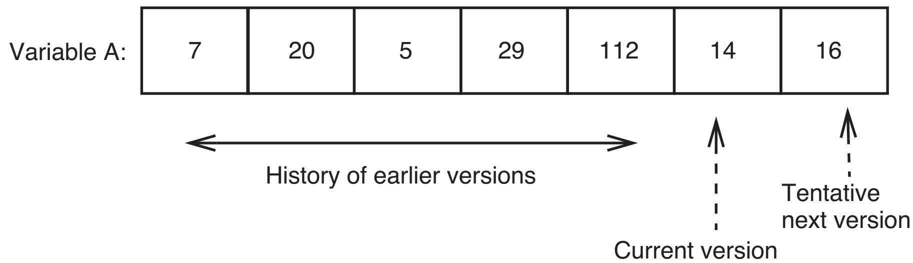


FIGURE 9.10

Version history of a variable in journal storage.

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 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

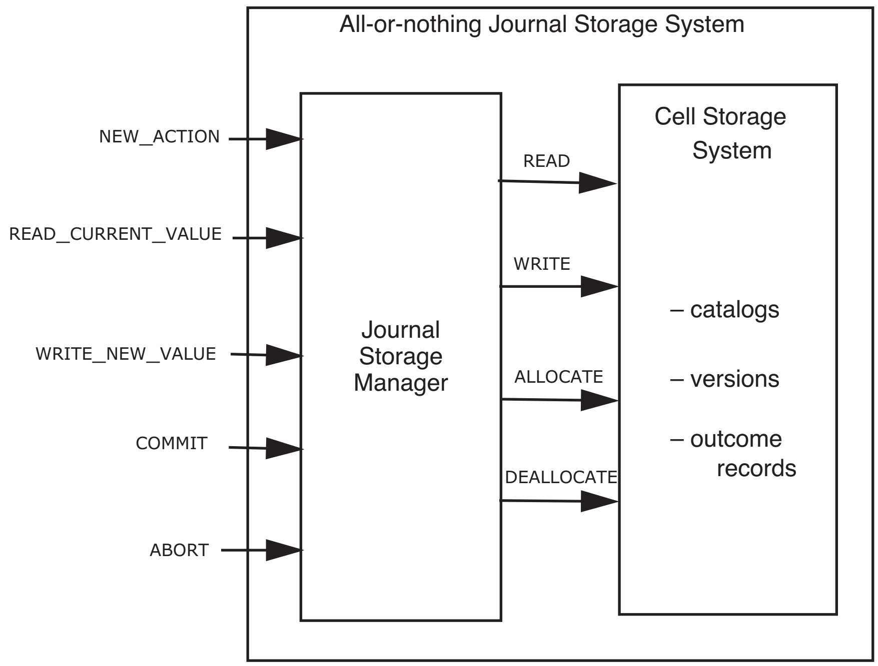


FIGURE 9.11

Interface to and internal organization of an all-or-nothing storage system based on version histories and journal storage.

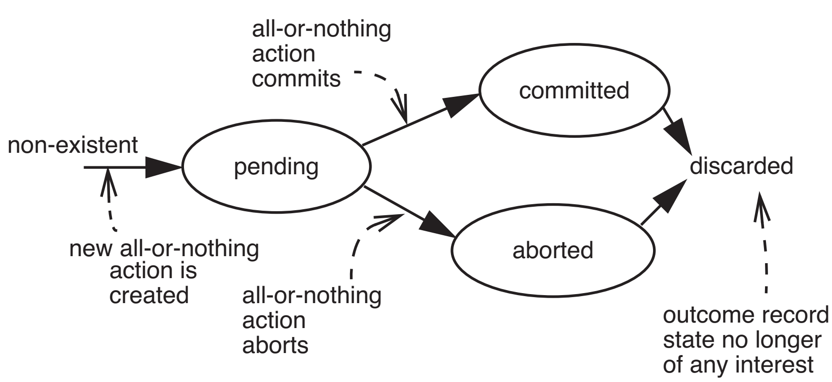


FIGURE 9.12

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.

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,

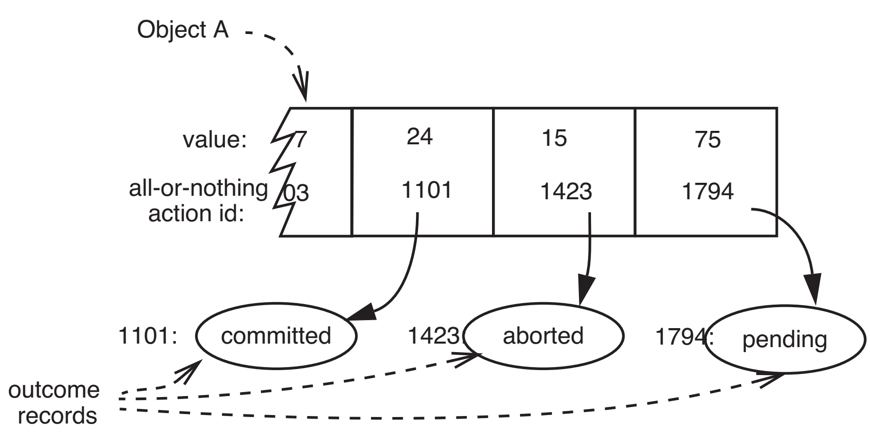


FIGURE 9.14

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.

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

10 procedure WRITE_NEW_VALUE (reference data_id, new_value, caller_id)
11     if caller_id.outcome_record.state = PENDING
12         append new version v to data_id
13         v.value ← new_value
14         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.

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 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 different procedure, named READ_MY_PENDING_VALUE, identical to READ_CURRENT_VALUE except for a different test on line 6, could do that.) Moreover if, for example, all-or-nothing action 99 crashes while partway through changing the values of nineteen different 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 before-or-after atomicity, but for the moment our only

```
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. Making the transfer procedure a before-or-after action because other threads might be updating the same accounts concurrently requires additional mechanism that is discussed later in this chapter.)

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 of Section 9.2.1, 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 Section 9.1.5 reprogrammed as an all-or-nothing (but not, for the moment, before-or-after) 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 before-or-after atomicity. Section 9.4 will explore methods of providing it.)

9.2.4 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 before-or-after atomicity in Section 9.4 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 database* 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

before-or-after 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.

9.3 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 describes.

9.3.1 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 will 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. The two steps, LOG and INSTALL, become a different implementation of the WRITE_NEW_VALUE interface of Figure 9.11. Figure 9.17 illustrates this two-step implementation.

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. 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*

---

* A hardware architect would say "…it *graduates* the change to cell storage". This text, somewhat arbitrarily, chooses to use the database management term "install" .

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.

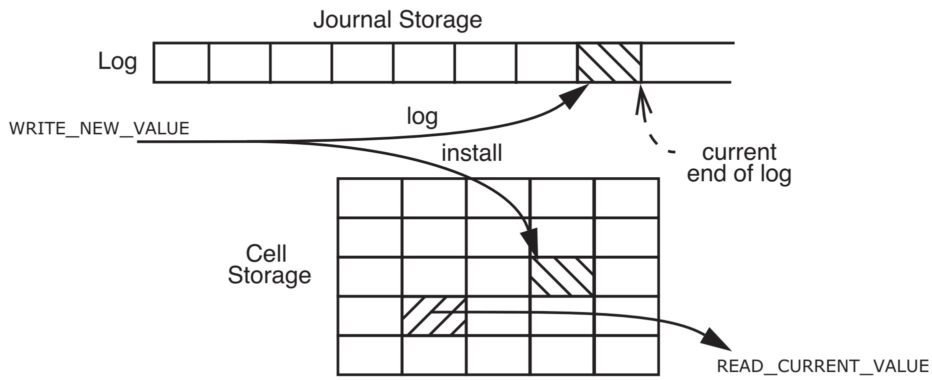


FIGURE 9.17

Logging for all-or-nothing atomicity. The application performs WRITE_NEW_VALUE by first appending a record of the new value to the log in journal storage, and then installing the new value in cell storage by overwriting. The application performs READ_CURRENT_VALUE by reading just from cell storage.

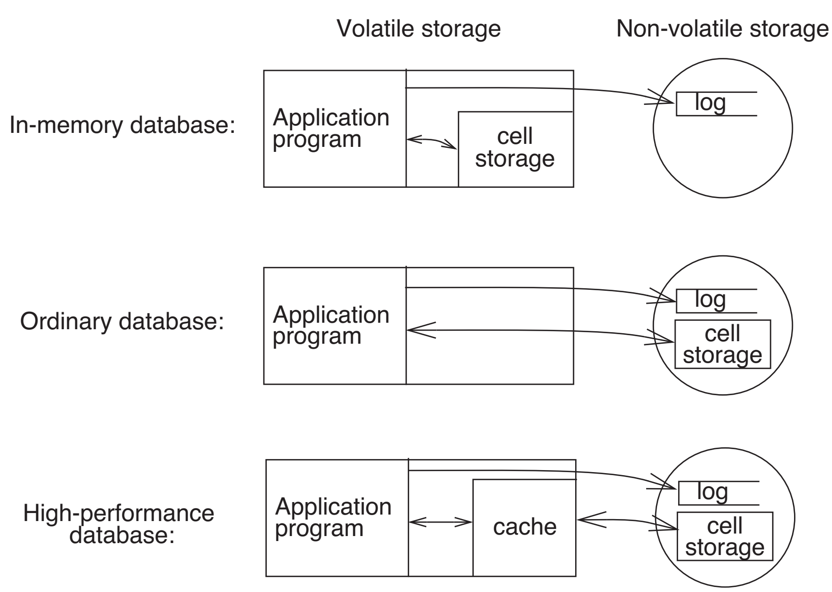


FIGURE 9.18

Three common logging configurations. Arrows show data flow as the application reads, logs, and installs data.

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.)

9.3.2 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 *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 will 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.

As we saw in Figure 9.17, LOG and INSTALL are the logging implementation of the WRITE_NEW_VALUE part of the interface of Figure 9.11, and READ_CURRENT_VALUE is simply a READ from cell storage. We also need a logging implementation of the remaining parts of the Figure 9.11 interface. The way to implement NEW_ACTION 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. To implement 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. As an example, Figure 9.19 shows our by now familiar TRANSFER action implemented with logging.

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 will see, some logging systems can combine, e.g., OUTCOME and END, or BEGIN with the first CHANGE.) Figure 9.20 shows examples of three log records, two typical CHANGE records of an all-or-nothing TRANSFER action, interleaved with the OUTCOME record of some other, perhaps completely unrelated, 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 a

```
procedure TRANSFER (debit_account, credit_account, amount)
    my_id ← LOG (BEGIN_TRANSACTION)
    dbvalue.old ← GET (debit_account)
    dbvalue.new ← dbvalue.old - amount
    crvalue.old ← GET (credit_account, my_id)
    crvalue.new ← crvalue.old + amount
    LOG (CHANGE, my_id,
        "PUT (debit_account, dbvalue.new)",              //redo action
        "PUT (debit_account, dbvalue.old)" )             //undo action
    LOG ( CHANGE, my_id,
        "PUT (credit_account, crvalue.new)"              //redo action
        "PUT (credit_account, crvalue.old)")             //undo action
    PUT (debit_account, dbvalue.new)                     // install
    PUT (credit_account, crvalue.new)                    // install
    if dbvalue.new > 0 then
        LOG ( OUTCOME, COMMIT, my_id)
    else
        LOG (OUTCOME, ABORT, my_id)
        signal("Action not allowed. Would make debit account negative.")
    LOG (END_TRANSACTION, my_id)
```

FIGURE 9.19

An all-or-nothing TRANSFER procedure, implemented with logging.

before-or-after atomicity concern, one of the places where all-or-nothing atomicity and before-or-after 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 proce-

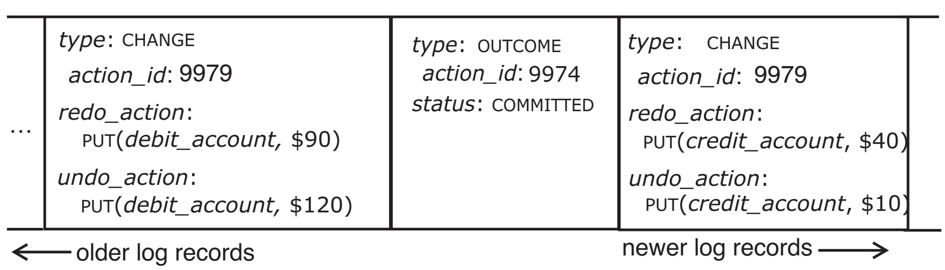


FIGURE 9.20

An example of a section of an atomicity log, showing two CHANGE records for a TRANSFER action that has action_id 9979 and the OUTCOME record of a different all-or-nothing action.

dure 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 9.21 illustrates.

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.

9.3.3 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

```
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)
```

FIGURE 9.21

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.

```
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 (outcome_record ← find (log_record.action_id) in winners)
            and (outcome_record.status = COMMITTED) then
            perform log_record.redo_action
```

FIGURE 9.22

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.

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 9.22 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.

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 9.21 will satisfy this requirement if the individual undo actions are themselves idempotent.

9.3.4 Other Logging Configurations: Non-Volatile Cell Storage

Placing cell storage in volatile memory is a *sweeping simplification* 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 place 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 for the moment assume that the multilevel memory manager implements a write-through cache. (Section 9.3.6, 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 temporarily 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 9.22. 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 9.23. Finally, the recovery 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

```
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.
```

FIGURE 9.23

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.

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 a blind write is 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.

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

```
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                  // 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.
```

FIGURE 9.24

An idempotent undo-only recovery procedure for rollback logging.

redo installs for any committed action that has logged its END record. A useful exercise is to modify the procedure of Figure 9.23 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 9.24. 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 know 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 database 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.

9.3.5 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 9.23 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 dealing with update activity that is concurrent with the writing of the snapshot. That challenge can be met either by preventing all updates for the duration of the snapshot or by applying more complex before-or-after atomicity techniques such as those described in later sections of this chapter.

9.3.6 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 before-or-after action to avoid getting tangled with 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.

9.4 Before-or-After 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 *before-or-after atomicity*. In this development we will provide *both* all-or-nothing atomicity *and* before-or-after 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 a composite action atomic with respect to concurrent threads—that is, making it a *before-or-after action*—requires further effort.

Recall that Section 9.1.5 defined the operation 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, yet maximize the performance that can be achieved by allowing concurrency.

In this Section 9.4 we explore three successively better before-or-after atomicity schemes, where "better" means that the scheme allows more concurrency. To illustrate the concepts we return to version histories, which allow a straightforward and compelling correctness argument for each scheme. Because version histories are rarely used in practice, in the following Section 9.5 we examine a somewhat different approach, locks, which are widely used because they can provide higher performance, but for which correctness arguments are more difficult.

9.4.1 Achieving Before-or-After Atomicity: 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 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 9.25 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

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

FIGURE 9.25

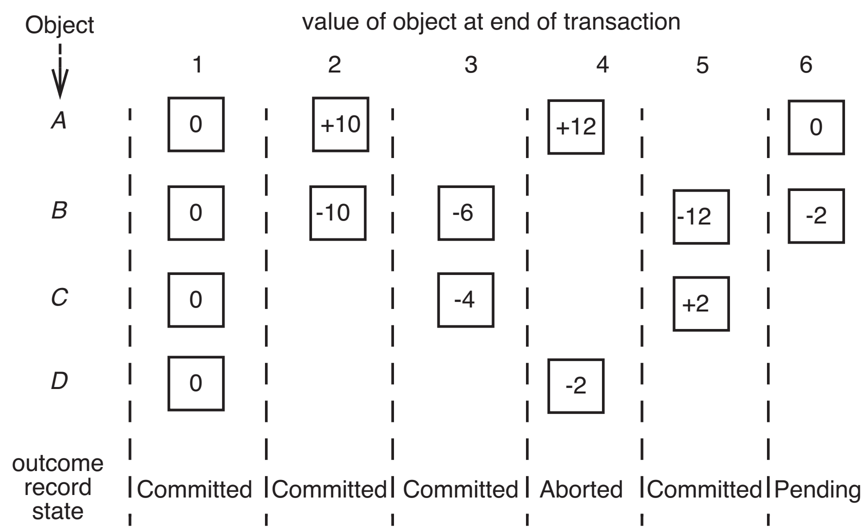
BEGIN_TRANSACTION with the simple serialization discipline to achieve before-or-after atomicity. In order that there be an *id* – 1 for every value of *id*, startup of the system must include creating a dummy transaction with *id* = 0 and *id.outcome_record.state* set to COMMITTED. Pseudocode for the procedure NEW_OUTCOME_RECORD appears in Figure 9.30.

order is a possible serial order of the various transactions, by definition simple serialization will produce transactions that are serialized and thus are correct before-or-after actions. Simple serialization trivially provides before-or-after atomicity, 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 before-or-after atomicity by being too conservative: it prevents 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 actions that need before-or-after atomicity 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 before-or-after atomicity 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 9.26 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



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 9.26

Version history of a banking system.

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 ensures correctness.

Figure 9.26 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.

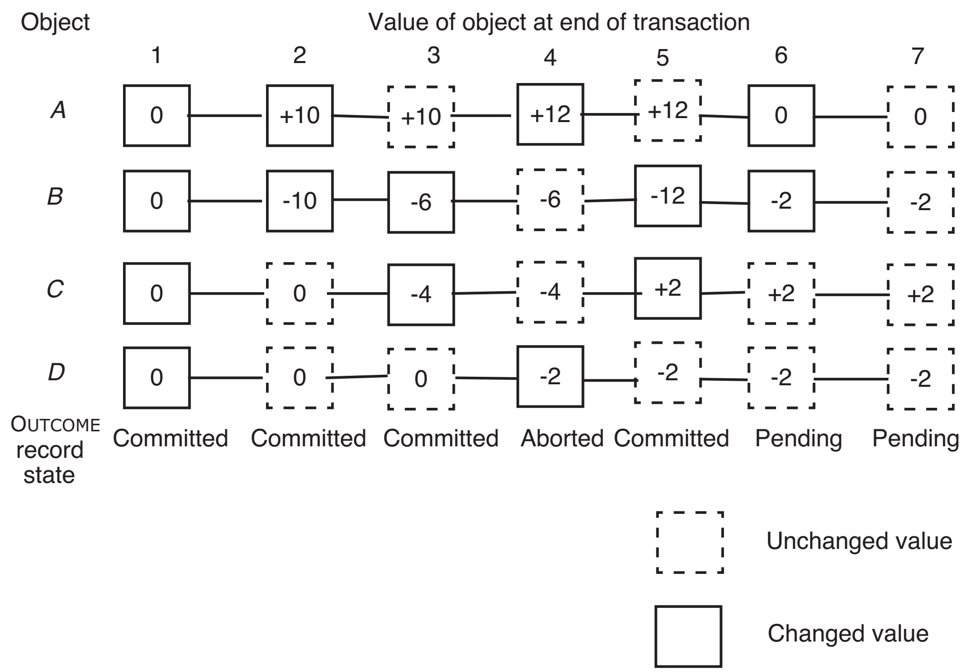


FIGURE 9.27

System state history with unchanged values shown.

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 9.26 represents the state history of the entire system in serialization order, but the slightly different representation of Figure 9.27 makes that state history more explicit. In Figure 9.27 it appears that each transaction has perversely created a new version of every 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 9.27, 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 9.27 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 9.27 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 ensure before-or-after atomicity with respect to other transactions if it can discover the dotted-versus-solid status of those version history boxes to its left. The observation also leads to a specific before-or-after atomicity discipline that will ensure correctness. We call this discipline *mark-point*.

9.4.2 The Mark-Point Discipline

Concurrent threads that invoke READ_CURRENT_VALUE as implemented in Figure 9.15 can not see a pending version of any variable. That observation is useful in designing a before-or-after atomicity 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 4–9 of Figure 9.28. 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 before-or-after atomicity 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 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 will 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 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 9.27, 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 9.29. 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 9.30, so that no transaction may begin execution until the preceding transaction either reports that it has reached its mark point or is no longer PENDING. Figure 9.30 also illustrates an implementation of MARK_POINT_ANNOUNCE. No changes are needed in procedures ABORT and COMMIT as shown in Figure 9.13, 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

```
procedure READ_CURRENT_VALUE (data_id, this_transaction_id)
    starting at end of data_id repeat until beginning
        v ← previous version of data_id
        last_modifier ← v.action_id
        if last_modifier ≥ this_transaction_id then skip v         // Keep searching
        wait until (last_modifier.outcome_record.state ≠ PENDING)
        if (last_modifier.outcome_record.state = COMMITTED)
            then return v.state
            else skip v                                            // Resume search
    signal ("Tried to read an uninitialized variable")
```

FIGURE 9.28

READ_CURRENT_VALUE for the mark-point 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 9.29

Mark-point discipline versions of NEW_VERSION and WRITE_VALUE.

```
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)         // Make this a before-or-after action.
    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 9.30

The procedures BEGIN_TRANSACTION, NEW_OUTCOME_RECORD, and MARK_POINT_ANNOUNCE for the mark-point 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.

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 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 before-or-after 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 before-or-after atomicity by bootstrapping from a more primitive before-or-after atomicity mechanism. As usual in bootstrapping, the idea is to reduce some general problem—here, that problem is to provide before-or-after atomicitiy for 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 9.30 must itself be a before-or-after action because it may be invoked concurrently by several different threads 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 before-or-after atomicity, NEW_OUTCOME_RECORD bootstraps from the TICKET procedure of Section 5.6.3 to obtain the next sequential serial number, and it uses ACQUIRE and RELEASE to make its initialization steps a before-or-after action. Those procedures in turn bootstrap from still lower-level before-or-after atomicity 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 9.31. 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 observation about this program is that most of the work of making actions before-or-after 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 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

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

FIGURE 9.31

An implementation of the funds transfer procedure that uses the mark point discipline to ensure that it is atomic both with respect to failure and with respect to concurrent activity.

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 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 other transactions to complete, 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 a before-or-after atomicity discipline. In Section 9.5 of this chapter we will encounter before-or-after atomicity 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 ensure 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 discipline reduces to the simple serialization 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 discipline to discover and report 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 9.29.

9.4.3 Optimistic Atomicity: Read-Capture (Advanced Topic)

Both the simple serialization and mark-point disciplines 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. In doing so, they also may prevent some concurrency that would have been harmless to correctness. 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 an interfering transaction and makaing it 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 9.27 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 rerun 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 9.32, which has the same version history as did Figure 9.27.

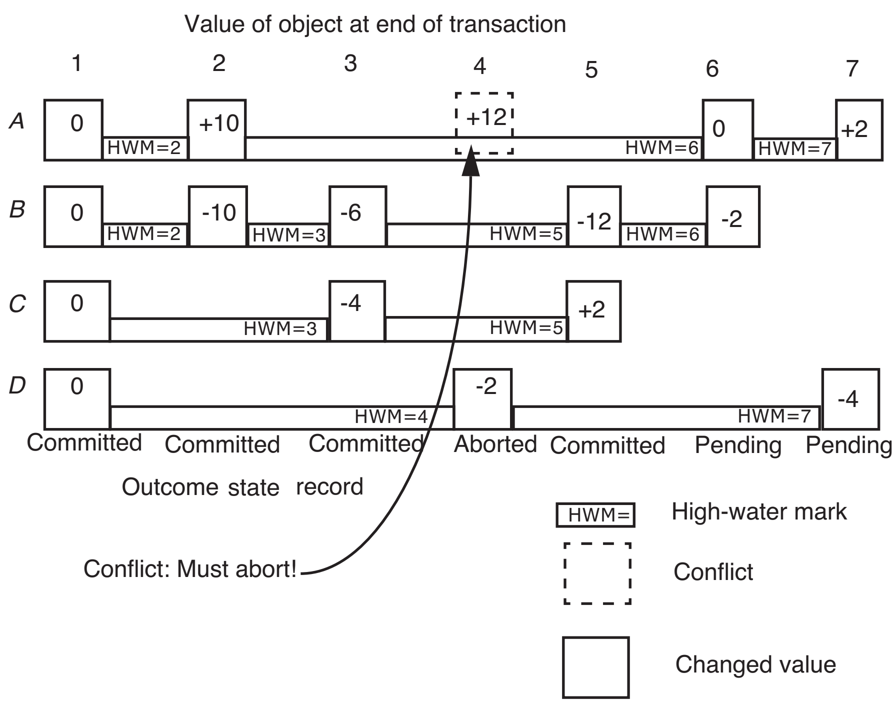


FIGURE 9.32

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.

The key property of read-capture is illustrated by an example in Figure 9.32. 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.

```
1  procedure READ_CURRENT_VALUE (reference data_id, value, caller_id)
2      starting at end of data_id repeat until beginning
3          v ← previous version of data_id
4          if v.action_id ≥ caller_id then skip v
5          examine v.action_id.outcome_record
6              if PENDING then
7                  WAIT for v.action_id to COMMIT or ABORT
8                  if COMMITTED then
9                      v.high_water_mark ← max(v.high_water_mark, caller_id)
10                     return v.value
11                 else skip v                          // Continue backward search
12     signal ("Tried to read an uninitialized variable!")

13 procedure NEW_VERSION (reference data_id, caller_id)
14     if (caller_id < data_id.high_water_mark)        // Conflict with later reader.
15     or (caller_id < (LATEST_VERSION[data_id].action_id))      // Blind write conflict.
16     then ABORT this transaction and terminate this thread
17     add new version v at end of data_id
18     v.value ← 0
19     v.action_id ← caller_id

20 procedure WRITE_VALUE (reference data_id, new_value, caller_id)
21     locate version v of data_id.history such that v.action_id = caller_id
22             (if not found, signal ("Tried to write without creating new version!"))
23     v.value ← new_value
```

FIGURE 9.33

Read-capture forms of READ_CURRENT_VALUE, NEW_VERSION, and WRITE_VALUE.

Read-capture is relatively easy to implement in a version history system. We start, as shown in Figure 9.33, by adding a new step (at line 9) 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 14), which checks the high-water mark for the object to be written, to see if any transaction with 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 15), 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 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 whether or not it actually works: is the result always the same as some serial ordering of the concurrent transactions? Because the read-capture 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 before-or-after atomicity 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.

9.4.4 Does Anyone Actually Use Version Histories for Before-or-After 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 Section 5.7, 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 9.29. As each instruction issues (which corresponds to BEGIN_TRANSACTION), it is assigned the next sequential slot in the reorder buffer. The slot is a map that maintains a correspondence between 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 9.34 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 of the mark-point or read-capture disciplines, 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 1.1.1].

The Oracle database management system offers several before-or-after atomicity methods, one of which it calls "serializable", though the label may be a bit misleading. This method uses a before-or-after atomicity 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

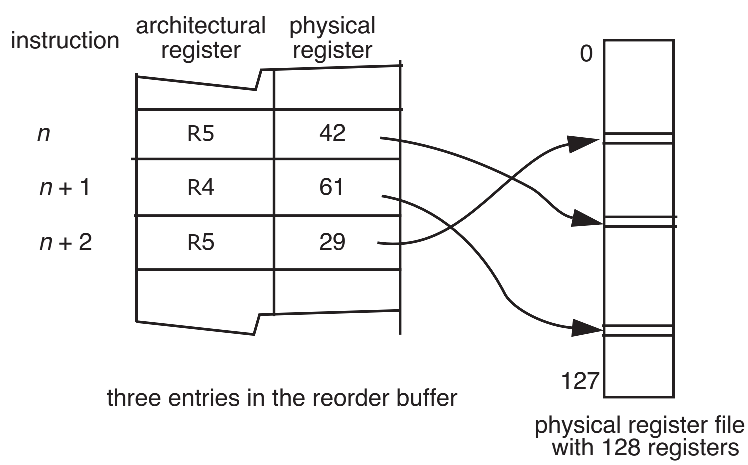


FIGURE 9.34

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.

history that, in certain situations, does not always ensure that concurrent transactions are correctly coordinated.

Another specialized variant implementation of version histories, known as *transactional memory*, is a discipline for creating atomic actions from arbitrary instruction sequences that make multiple references to primary memory. Transactional memory was first suggested in 1993 and with widespread availability of multicore processors, has become the subject of quite a bit of recent research interest because it allows the application programmer to use concurrent threads without having to deal with locks. The discipline is to mark the beginning of an instruction sequence that is to be atomic with a "begin transaction" instruction, direct all ensuing STORE instructions to a hidden copy of the data that concurrent threads cannot read, and at end of the sequence check to see that nothing read or written during the sequence was modified by some other transaction that committed first. If the check finds no such earlier modifications, the system commits the transaction by exposing the hidden copies to concurrent threads; otherwise it discards the hidden copies and the transaction aborts. Because it defers all discovery of interference to the commit point this discipline is even more optimistic than the read-capture discipline described in Section 9.4.3 above, so it is most useful in situations where interference between concurrent threads is possible but unlikely. Transactional memory has been experimentally implemented in both hardware and software. Hardware implementations typically involve tinkering with either a cache or a reorder buffer to make it defer writing hidden copies back to primary memory until commit time, while software implementations create hidden copies of changed variables somewhere else in primary memory. As with instruction renaming, this description of transactional memory is somewhat oversimplified, and the interested reader should consult the literature for fuller explanations.

Other software implementations of version histories for before-or-after atomicity 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. Before-or-after atomicity by using locks systematically is the subject of the next section of this chapter.

9.5 Before-or-After Atomicity II: Pragmatics

The previous section showed that a version history system that provides all-or-nothing atomicity can be extended to also provide before-or-after 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 before-or-after atomicity. When a system uses logs for all-or-nothing atomicity, it usually adopts the mechanism introduced in Chapter 5—*locks*—for before-or-after 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 before-or-after atomicity.

9.5.1 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 will succeed; the others must find the lock already acquired. In Section 9.5.4 we will 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 create actions that do not have the intended before-or-after property. Such errors can open the door 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 coordination 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 have the before-or-after property.
- 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 coordinate transactions (for an example, see exercise 9.5). We examine three disciplines that succeed. Each allows more concurrency than its predecessor, though even the best one is not capable of guaranteeing that concurrency is maximized.

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

```
begin_transaction
ACQUIRE (System.lock)
...
```

and every transaction must end with

```
...
RELEASE (System.lock)
end_transaction
```

A system can even enforce this discipline by including the ACQUIRE and RELEASE steps in the code sequence generated for **begin_transaction** and **end_transaction**, independent of whether the result was COMMIT or ABORT. Any programmer who creates a new transaction then has a guarantee that it will run either before or after any other transactions.

The systemwide lock discipline allows only one transaction to execute at a time. It serializes potentially concurrent transactions in the order that they call ACQUIRE. The systemwide lock discipline is in all respects identical to the simple serialization discipline of Section 9.4. 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 systemwide lock invocation

```
ACQUIRE (System.lock)
...
RELEASE (System.lock)
```

are actually just two implementations of the same idea.

As with simple serialization, systemwide locking restricts concurrency in cases where it doesn't need to because it locks all data touched by every transaction. For example, if systemwide locking were applied to the funds TRANSFER program of Figure 9.16, only one transfer could occur at a time, even though any individual transfer involves only two out of perhaps several million accounts, so there would be many opportunities for concurrent, non-interfering transfers. 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-offs in gaining concurrency are first, that when there is more than one lock, more time is spent acquiring and releasing locks and second, correctness arguments become more complex. One hopes that the performance gain from concurrency exceeds the cost of acquiring and releasing the multiple locks. Fortunately, there are at least two other disciplines for which correctness arguments are feasible, *simple locking* and *two-phase locking*.

9.5.2 Simple Locking

The second locking discipline, known as *simple locking,* is similar in spirit to, though not quite identical with, the mark-point discipline. The simple locking 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 can enforce simple locking by requiring that each transaction supply its intended lock set as an argument to the **begin_transaction** 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 changes, 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 coordinates concurrent transactions. We can make that claim using a line of argument analogous to the one used for correctness of the mark-point 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, and by induction to any transaction earlier 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, 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 ensures that this transaction runs completely after the preceding one and completely before the next one. Concurrent transactions will produce results as if they had been serialized in the order that they reached their lock points.

As with the mark-point discipline, simple locking can miss 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 the mark-point discipline 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 that an application *might* need to read is larger than the set for which it eventually *does* read, the simple locking discipline can interfere with opportunities for concurrency. On the other hand, when the transaction is straightforward (such as the TRANSFER transaction of Figure 9.16, which needs to lock only two records, both of which are known at the outset) simple locking can be effective.

9.5.3 Two-Phase Locking

The third locking discipline, called *two-phase locking,* like the read-capture discipline, avoids the requirement that a transaction must know in advance which locks to acquire. Two-phase locking is widely used, but it is harder to argue that it is correct. 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 an object that it only reads any time after it reaches its lock point *if* it will never need to read 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 orders 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 before-or-after atomicity. 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 before-or-after atomicity 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].

The two-phase locking discipline can potentially allow more concurrency than the simple locking discipline, but it still unnecessarily blocks certain serializable, and therefore correct, action orderings. For example, suppose transaction T_1 reads *X* and writes *Y*, while transaction T_2 just does a (blind) write to *Y*. Because the lock sets of T_1 and T_2 intersect at variable *Y*, the two-phase locking discipline will force transaction T_2 to run either completely before or completely after T_1. But the sequence

T_1: READ *X*
T_2: WRITE *Y*
T_1: WRITE *Y*

in which the write of T_2 occurs between the two steps of T_1, yields the same result as running T_2 completely before T_1, so the result is always correct, even though this sequence would be prevented by two-phase locking. Disciplines that allow all possible concurrency while at the same time ensuring before-or-after atomicity are quite difficult to devise. (Theorists identify the problem as NP-complete.)

There are two interactions between locks and logs that require some thought: (1) individual transactions that abort, and (2) 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. For purposes of before-or-after atomicity 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 coordinate 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 9.23 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 must have been non-overlapping, those particular actions can safely be redone or undone without concern for before-or-after atomicity during recovery. Put another way, 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 database 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.

9.5.4 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 9.3.6 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 ensure 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 before-or-after atomicity 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 before-or-after atomicity, it would be advisable to study carefully the book by Gray and Reuter, as well as existing systems that implement similar enhancements.

9.5.5 Deadlock; Making Progress

Section 5.2.5 of Chapter 5 introduced the emergent problem of *deadlock*, the wait-for graph as a way of analyzing deadlock, and lock ordering as a way of preventing deadlock. With transactions and the ability to undo individual actions or even abort a transaction completely we now have more tools available to deal with deadlock, so it is worth revisiting that discussion.

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 ensure 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 ensure *correctness*, in the sense that no wrong answers ever come out, but it does not ensure *progress*—no answers may come out at all.

As with methods for concurrency control, methods for coping with deadlock can also be described as pessimistic or optimistic. Pessimistic methods take *a priori* action to prevent deadlocks from happening. Optimistic methods allow concurrent threads to proceed, detect deadlocks if they happen, and then take action to fix things up. Here are some of the most popular methods:

1. *Lock ordering* (pessimistic). 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 locks they need before acquiring the first one.

2. *Backing out* (optimistic): An elegant strategy devised by Andre Bensoussan in 1966 allows a transaction to acquire locks in any order, but if it encounters an already-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. *Timer expiration* (optimistic). 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.

4. *Cycle detection* (optimistic). Maintain, in the lock manager, a wait-for graph (as described in Section 5.2.5) that shows which transactions have acquired which locks and which transactions are waiting for which locks. Whenever another transaction tries to acquire a lock, finds it is already locked, and proposes to wait, the lock manager examines the graph to see if waiting would produce a cycle, and thus a deadlock. If it would, the lock manager selects some cycle member to be a 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, backing out, 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 none of the deadlock detection methods 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 emergent property that Chapter 5 called *livelock*, since formally no deadlock ever occurs 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[on-line]: *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.

9.6 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 will 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

```
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)

procedure MONTH_END_INTEREST:()
    for A ← each customer_account do
        PAY_INTEREST (A)
```

FIGURE 9.35

An example of two procedures, one of which calls the other, yet each should be individually atomic.

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 quite similar to, but not quite the same as, the *dilemma of the two generals*.

9.6.1 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 Section 9.1.5 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 9.35. 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.

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 run either before or after any concurrent TRANSFER transactions), but it must be able to do multiple TRANSFERS itself, each time it invokes PAY_INTEREST, and its own possibly concurrent transfer actions must be before-or-after actions, 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 in Section 9.4.2) 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 will 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 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 in Section 9.4.2, doesn't distinguish between reads arising within the same group of nested transactions and reads from some completely unrelated transaction. Figure 9.36 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

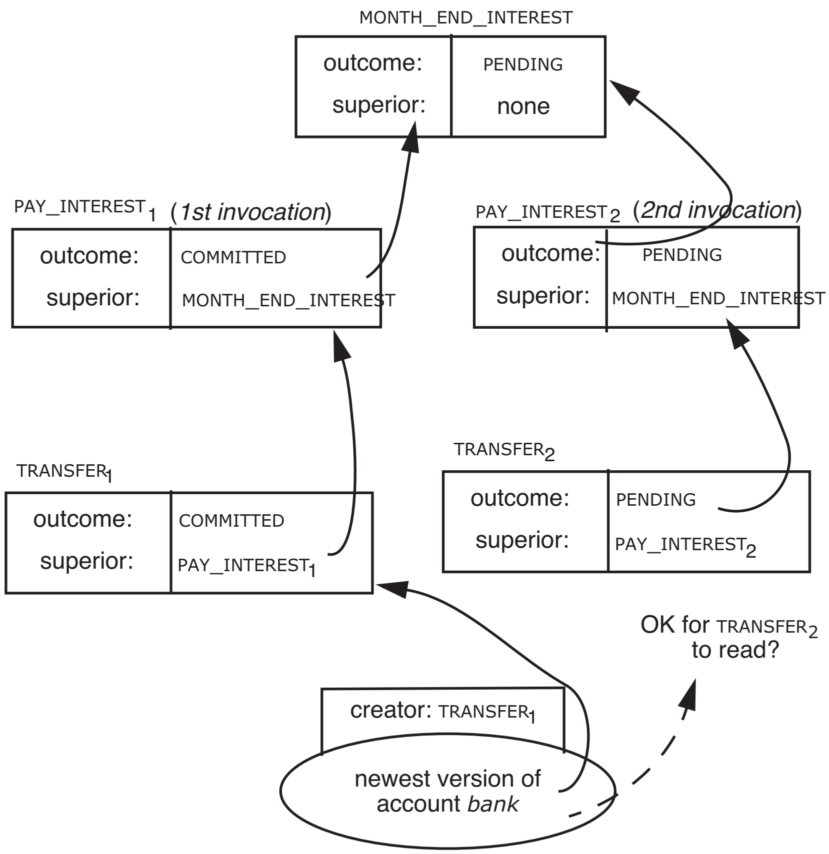


FIGURE 9.36

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.

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 before-or-after atomicity 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 9.36, MONTH_END_INTEREST might be running the two (or more) invocations of PAY_INTEREST concurrently. Each invocation will call 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.

9.6.2 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 9.36 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.

9.6.3 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[on-line] we learned how to design protocols such as RPC with a persistent sender to ensure at-least-once execution and duplicate suppression to ensure at-most-once execution. Unfortunately, neither of these two assurances is exactly what is needed to ensure 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 before-or-after 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 before-or-after atomicity, 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 ensuring that they will act simultaneously. Instead, Alice can only be certain 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 9.37: 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 acknowledgment message from the worker sites to the coordinator. The intent is to collect a complete set of acknowledgment messages—the coordinator persistently sends completion messages until every site acknowledges. Once all acknowledgments 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 acknowledgment 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 acknowledgment of aborted transactions, and discard the outcome record after all these acknowledgments are in. This protocol that leads to discarding an outcome record is identical to the protocol described in Chapter 7[on-line] 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

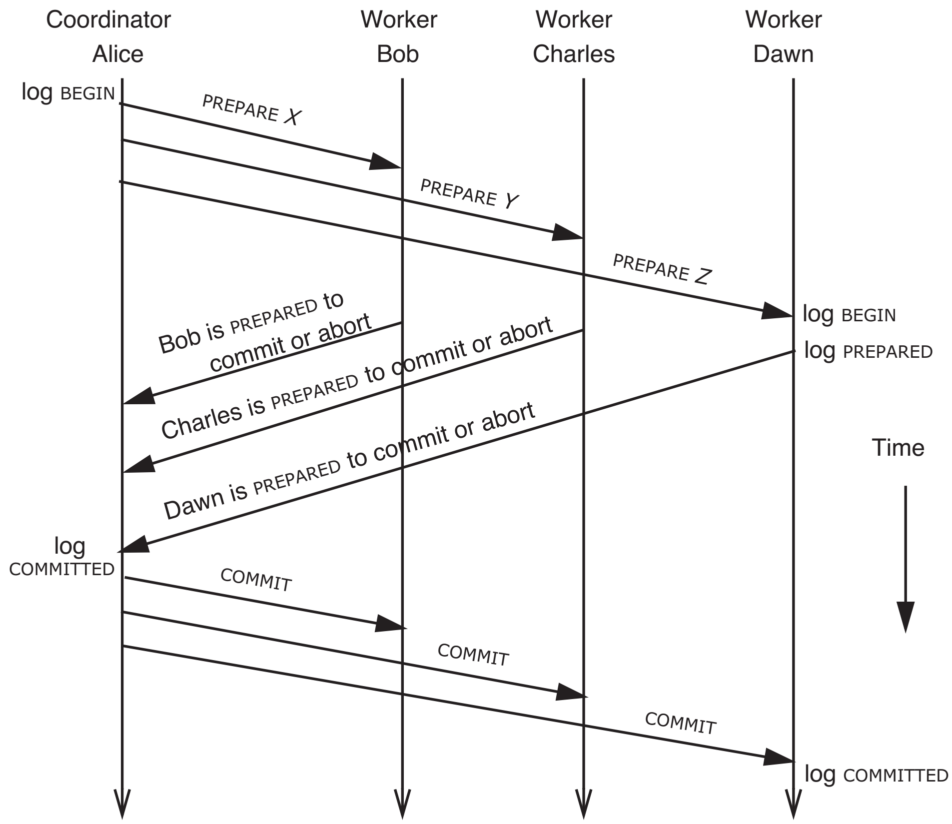


FIGURE 9.37

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.

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.*

9.6.4 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.
```

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

* 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.

general received the confirmation. This hesitation on the part of the second general suggests that the first general should send back an acknowledgment 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 acknowledgment may be lost and the second general, not receiving the acknowledgment, 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[on-line] called an intolerable error.) Similarly, the second general sends many runners each carrying a copy of either the "Yes" or the "No" acknowledgment. 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 ensure 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 ensures correctness.

By a similar line of reasoning, there is no way to ensure 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.

9.7 A More Complete Model of Disk Failure (Advanced Topic)

Section 9.2 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 section 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.

9.7.1 Storage that is Both All-or-Nothing and Durable

In Chapter 8[on-line] 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[on-line]. 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[on-line] 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 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 ensure 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 9.38 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 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[on-line], 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.

```
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 Td 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)
```

D_0: $\boxed{data_0}$ D_1: $\boxed{data_1}$

FIGURE 9.38

Data arrangement and algorithms to implement all-or-nothing durable storage on top of the careful storage layer of Figure 8.12.

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 [Suggestions for Further Reading 1.8.7]. 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.

9.8 Case Studies: Machine Language Atomicity

9.8.1 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 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.

9.8.2 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.

9.8.3 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.

Exercises

9.1 *Locking up humanities*: The registrar's office is upgrading its scheduling program for limited-enrollment humanities subjects. The plan is to make it multithreaded, but there is concern that having multiple threads trying to update the database at the same time could cause trouble. The program originally had just two operations:

status ← REGISTER (*subject_name*)
DROP (*subject_name*)

where *subject_name* was a string such as "21W471". The REGISTER procedure checked to see if there is any space left in the subject, and if there was, it incremented the class size by one and returned the status value ZERO. If there was no space, it did not change the class size; instead it returned the status value –1. (This is a primitive registration system—it just keeps counts!)

As part of the upgrade, *subject_name* has been changed to a two-component structure:

structure *subject*
    string *subject_name*
    lock *slock*

and the registrar is now wondering where to apply the locking primitives,

ACQUIRE (*subject.slock*)
RELEASE (*subject.slock*)

Here is a typical application program, which registers the caller for two humanities subjects, *hx* and *hy*:

```
procedure REGISTER_TWO (hx, hy)
    status ← REGISTER (hx)
    if status = 0 then
        status ← REGISTER (hy)
        if status = -1 then
            DROP (hx)
    return status;
```

9.1a. The goal is that the entire procedure REGISTER_TWO should have the before-or-after property. Add calls for ACQUIRE and RELEASE to the REGISTER_TWO procedure that obey the *simple locking protocol*.

9.1b. Add calls to ACQUIRE and RELEASE that obey the *two-phase locking protocol*, and in addition postpone all ACQUIREs as late as possible and do all RELEASEs as early as possible.

Louis Reasoner has come up with a suggestion that he thinks could simplify the job of programmers creating application programs such as REGISTER_TWO. His idea is to revise the two programs REGISTER and DROP by having them do the ACQUIRE and RELEASE internally. That is, the procedure:

```
procedure REGISTER (subject)
    { current code }
    return status
```

would become instead:

```
procedure REGISTER (subject)
    ACQUIRE (subject.slock)
    { current code }
    RELEASE (subject.slock)
    return status
```

9.1c. As usual, Louis has misunderstood some aspect of the problem. Give a brief explanation of what is wrong with this idea.

1995–3–2a…c

9.2 Ben and Alyssa are debating a fine point regarding version history transaction disciplines and would appreciate your help. Ben says that under the mark point transaction discipline, every transaction should call MARK_POINT_ANNOUNCE as soon as possible, or else the discipline won't work. Alyssa claims that everything will come out correct even if no transaction calls MARK_POINT_ANNOUNCE. Who is right?

2006-0-1

9.3 Ben and Alyssa are debating another fine point about the way that the version history transaction discipline bootstraps. The version of NEW_OUTCOME_RECORD given in the text uses TICKET as well as ACQUIRE and RELEASE. Alyssa says this is overkill—it should be possible to correctly coordinate NEW_OUTCOME_RECORD using just ACQUIRE and RELEASE. Modify the pseudocode of Figure 9.30 to create a version of NEW_OUTCOME_RECORD that doesn't need the ticket primitive.

9.4 You have been hired by Many-MIPS corporation to help design a new 32-register RISC processor that is to have six-way multiple instruction issue. Your job is to coordinate the interaction among the six arithmetic-logic units (ALUs) that will be running concurrently. Recalling the discussion of coordination, you realize that the first thing you must do is decide what constitutes "correct" coordination for a multiple-instruction-issue system. Correct coordination for concurrent operations on a database was said to be:

No matter in what order things are actually calculated, the final result is always guaranteed to be one that could have been obtained by some sequential ordering of the concurrent operations.

You have two goals: (1) maximum performance, and (2) not surprising a programmer who wrote a program expecting it to be executed on a single-instruction-issue machine.

Identify the best coordination correctness criterion for your problem.

A. Multiple instruction issue must be restricted to sequences of instructions that have non-overlapping register sets.

B. No matter in what order things are actually calculated, the final result is always guaranteed to be one that could have been obtained by some sequential ordering of the instructions that were issued in parallel.

C. No matter in what order things are actually calculated, the final result is always guaranteed to be the one that would have been obtained by the original ordering of the instructions that were issued in parallel.

D. The final result must be obtained by carrying out the operations in the order specified by the original program.

E. No matter in what order things are actually calculated, the final result is always guaranteed to be one that could have been obtained by some set of instructions carried out sequentially.

F. The six ALUs do not require any coordination.

1997–0–02

9.5 In 1968, IBM introduced the Information Management System (IMS) and it soon became one of the most widely used database management systems in the world. In fact, IMS is still in use today. At the time of introduction IMS used a before-or-after atomicity protocol consisting of the following two rules:

- A transaction may read only data that has been written by previously committed transactions.
- A transaction must acquire a lock for every data item that it will write.

Consider the following two transactions, which, for the interleaving shown, both adhere to the protocol:

```
    BEGIN (t1);          BEGIN (t2)
    ACQUIRE (y.lock)
    temp1 ← x
                         ACQUIRE (x.lock)
                         temp2 ← y
                         x ← temp2
    y ← temp1
    COMMIT (t1)
                         COMMIT (t2)
```

Previously committed transactions had set $x \leftarrow 3$ and $y \leftarrow 4$.

9.5a. After both transactions complete, what are the values of *x* and *y*? In what sense is this answer wrong?

1982–3–3a

9.5b. In the mid-1970's, this flaw was noticed, and the before-or-after atomicity protocol was replaced with a better one, despite a lack of complaints from customers. Explain why customers may not have complained about the flaw.

1982–3–3b

9.6 A system that attempts to make actions all-or-nothing writes the following type of records to a log maintained on non-volatile storage:

- <STARTED *i*> — action *i* starts.
- <*i*, *x*, *old*, *new*> — action *i* writes the value *new* over the value *old* for the variable *x*.
- <COMMITTED *i*> — action *i* commits.
- <ABORTED *i*> — action *i* aborts.
- <CHECKPOINT *i*, *j*,…> — At this checkpoint, actions *i*, *j*,… are pending.

Actions start in numerical order. A crash occurs, and the recovery procedure finds the following log records starting with the last checkpoint:

```
<CHECKPOINT 17, 51, 52>
<STARTED 53>
<STARTED 54>
<53, y, 5, 6>
<53, x, 5, 9>
<COMMITTED 53>
<54, y, 6, 4>
<STARTED 55>
<55, z, 3, 4>
<ABORTED 17>
<51, q, 1, 9>
<STARTED 56>
<55, y, 4, 3>
<COMMITTED 54>
<55, y, 3, 7>
<COMMITTED 51>
<STARTED 57>
<56, x, 9, 2>
<56, w, 0, 1>
<COMMITTED 56>
<57, u, 2, 1>
****************** crash happened here **************
```

9.6a. Assume that the system is using a rollback recovery procedure. How much farther back in the log should the recovery procedure scan?

9.6b. Assume that the system is using a roll-forward recovery procedure. How much farther back in the log should the recovery procedure scan?

9.6c. Which operations mentioned in this part of the log are winners and which are losers?

9.6d. What are the values of x and y immediately after the recovery procedure finishes? Why?

1994–3–3

9.7 The log of exercise 9.6 contains (perhaps ambiguous) evidence that someone didn't follow coordination rules. What is that evidence?

1994–3–4

9.8 Roll-forward recovery requires writing the commit (or abort) record to the log *before* doing any installs to cell storage. Identify the best reason for this requirement.

A. So that the recovery manager will know what to undo.
B. So that the recovery manager will know what to redo.
C. Because the log is less likely to fail than the cell storage.
D. To minimize the number of disk seeks required.

1994–3–5

9.9 Two-phase locking within transactions ensures that

A. No deadlocks will occur.
B. Results will correspond to some serial execution of the transactions.
C. Resources will be locked for the minimum possible interval.
D. Neither gas nor liquid will escape.
E. Transactions will succeed even if one lock attempt fails.

1997–3–03

9.10 Pat, Diane, and Quincy are having trouble using e-mail to schedule meetings. Pat suggests that they take inspiration from the 2-phase commit protocol.

9.10a. Which of the following protocols most closely resembles 2-phase commit?

```
I.  a. Pat requests everyone's schedule openings.
    b. Everyone replies with a list but does not guarantee to hold all the times available.
    c. Pat inspects the lists and looks for an open time.
       If there is a time,
          Pat chooses a meeting time and sends it to everyone.
       Otherwise
          Pat sends a message canceling the meeting.

II. a–c, as in protocol I.
    d. Everyone, if they received the second message,
          acknowledge receipt.
       Otherwise
          send a message to Pat asking what happened.

III a–c, as in protocol I.
    d. Everyone, if their calendar is still open at the chosen time
          Send Pat an acknowledgment.
       Otherwise
          Send Pat apologies.
    e. Pat collects the acknowledgments. If all are positive
          Send a message to everyone saying the meeting is ON.
       Otherwise
          Send a message to everyone saying the meeting is OFF.
    f. Everyone, if they received the ON/OFF message,
          acknowledge receipt.
       Otherwise
          send a message to Pat asking what happened.

IV. a–f, as in protocol III.
    g. Pat sends a message telling everyone that everyone has confirmed.
    h. Everyone acknowledges the confirmation.
```

9.10b. For the protocol you selected, which step commits the meeting time?

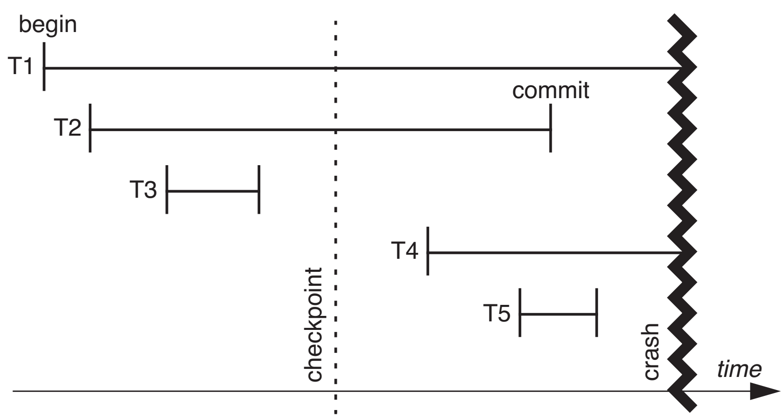
1994–3–7

9.11 Alyssa P. Hacker needs a transaction processing system for updating information about her collection of 97 cockroaches.*

9.11a. In her first design, Alyssa stores the database on disk. When a transaction commits, it simply goes to the disk and writes its changes in place over the old data. What are the major problems with Alyssa's system?

9.11b. *I*n Alyssa's second design, the *only* structure she keeps on disk is a log, with a reference copy of all data in volatile RAM. The log records every change made to the database, along with the transaction which the change was a part of. Commit records, also stored in the log, indicate when a transaction commits. When the system crashes and recovers, it replays the log, redoing each committed transaction, to reconstruct the reference copy in RAM. What are the disadvantages of Alyssa's second design?

To speed things up, Alyssa makes an occasional checkpoint of her database. To checkpoint, Alyssa just writes the entire state of the database into the log. When the system crashes, she starts from the last checkpointed state, and then redoes or undoes some transactions to restore her database. Now consider the five transactions in the illustration:



Transactions T2, T3, and T5 committed before the crash, but T1 and T4 were still pending.

9.11c. When the system recovers, after the checkpointed state is loaded, some transactions will need to be undone or redone using the log. For each transaction,

* Credit for developing exercise 9.11 goes to Eddie Kohler.

mark off in the table whether that transaction needs to be undone, redone, or neither.

	Undone	Redone	Neither
T1			
T2			
T3			
T4			
T5			

9.11d. Now, assume that transactions T2 and T3 were actually *nested* transactions: T2 was nested in T1, and T3 was nested in T2. Again, fill in the table

	Undone	Redone	Neither
T1			
T2			
T3			
T4			
T5			

1996–3–3

9.12 Alice is acting as the coordinator for Bob and Charles in a two-phase commit protocol. Here is a log of the messages that pass among them:

1	Alice ⇒ Bob:	please do X
2	Alice ⇒ Charles:	please do Y
3	Bob ⇒ Alice:	done with X
4	Charles ⇒ Alice:	done with Y
5	Alice ⇒ Bob:	PREPARE to commit or abort
6	Alice ⇒ Charles:	PREPARE to commit or abort
7	Bob ⇒ Alice:	PREPARED
8	Charles ⇒ Alice:	PREPARED
9	Alice ⇒ Bob:	COMMIT
10	Alice ⇒ Charles:	COMMIT

At which points in this sequence is it OK for Bob to abort his part of the transaction?

A. After Bob receives message 1 but before he sends message 3.
B. After Bob sends message 3 but before he receives message 5.
C. After Bob receives message 5 but before he sends message 7.
D. After Bob sends message 7 but before he receives message 9.
E. After Bob receives message 9.

2008–3–11

Additional exercises relating to Chapter 9 can be found in problem sets *29* through *40*.

Glossary for Chapter 9

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*. [Ch. 9]

all-or-nothing atomicity—A property of a multistep action that if an anticipated failure occurs during the steps of the action, the effect of the action from the point of view of its invoker is either never to have started or else to have been accomplished completely. Compare with *before-or-after atomicity* and *atomic*. [Ch. 9]

archive—A record, usually kept in the form of a log, of old data values, for auditing, recovery from application mistakes, or historical interest. [Ch. 9]

atomic (adj.); **atomicity** (n.)—A property of a multistep action that there be no evidence that it is composite above the layer that implements it. An atomic action can be before-or-after, which means that its effect is as if it occurred either completely before or completely after any other before-or-after action. An atomic action can also be all-or-nothing, which means that if an anticipated failure occurs during the action, the effect of the action as seen by higher layers is either never to have started or else to have completed successfully. An atomic action that is *both* all-or-nothing and before-or-after is known as a *transaction*. [Ch. 9]

atomic storage—Cell storage for which a multicell 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 in which multicell PUTs are not atomic are said to be subject to *write tearing*. [Ch. 9]

before-or-after atomicity—A property of concurrent actions: Concurrent actions are before-or-after actions if their effect from the point of view of their invokers is the same as if the actions occurred either completely before or completely after one another. One consequence is that concurrent before-or-after software actions cannot discover the composite nature of one another (that is, one action cannot tell that another has multiple steps). A consequence in the case of hardware is that concurrent before-or-after WRITEs to the same memory cell will be performed in some order, so there is no danger that the cell will end up containing, for example, the OR of several WRITE values. The database literature uses the words "isolation" and "serializable", the operating system literature uses the words "mutual exclusion" and "critical section", and the computer architecture literature uses the unqualified word "atomicity" for this concept. Compare with *all-or-nothing atomicity* and *atomic*. [Ch. 9]

blind write—An update to a data value *X* by a transaction that did not previously read *X*. [Ch. 9]

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*. [Ch. 9]

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. [Ch. 9]

close-to-open consistency—A consistency model for file operations. When a thread opens a file and performs several write operations, all of the modifications weill be visible to concurrent threads only after the first thread closes the file. [Ch. 4]

coheerence—See *read/write coherence* or *cache coherence*.

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 *compensate* and *abort*. [Ch. 9]

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. [Ch. 9]

do action—(n.) Term used in some systems for a *redo action*. [Ch. 9]

exponential random backoff—A form of *exponential backoff* in which an action that repeatedly encounters interference repeatedly doubles (or, more generally, multiplies by a constant greater than one) the size of an interval from which it randomly chooses its next delay before retrying. The intent is that by randomly changing the timing relative to other, interfering actions, the interference will not recur. [Ch. 9]

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. [Ch. 9]

install—In a system that uses logs to achieve all-or-nothing atomicity, to write data to cell storage. [Ch. 9]

journal storage—Storage in which a WRITE or PUT appends a new value, rather than overwriting a previously stored value. Compare with *cell storage*. [Ch. 9]

lock point—In a system that provides before-or-after atomicity by locking, the first instant in a before-or-after action when every lock that will ever be in its lock set has been acquired. [Ch. 9]

lock set—The collection of all locks acquired during the execution of a before-or-after action. [Ch. 9]

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 a record to a log. [Ch. 9]

logical locking—Locking of higher-layer data objects such as records or fields of a database. Compare with *physical locking*. [Ch. 9]

mark point—1. (adj.) An atomicity-assuring discipline in which each newly created 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 action has marked all of the variables it intends to modify. [Ch. 9]

optimistic concurrency control—A concurrency control scheme that allows concurrent threads to proceed even though a risk exists that they will interfere with each other, with the plan of detecting whether 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*. [Ch. 9]

page fault—See *missing-page exception*.

pair-and-spare—See *pair-and-compare*.

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. [Ch. 9]

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*. [Ch. 9]

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*. [Ch. 9]

prepaging—An optimization for a multilevel memory manager in which the manager predicts which pages might be needed and brings them into the primary memory before the application demands them. Compare with *demand algorithm*.

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. [Ch. 9]

presented load—See *offered load*.

progress—A desirable guarantee provided by an atomicity-assuring mechanism: that despite potential interference from concurrency some useful work will be done. An example of such a guarantee is that the atomicity-assuring 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, whereas liveness is an assurance that the system will eventually reach that goal. [Ch. 9]

redo action—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.*) [Ch. 9]

roll-forward recovery—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. Also known as *redo logging*. Compare with *rollback recovery.* [Ch. 9]

rollback recovery—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. Also known as *undo logging*. Compare with *roll-forward recovery.* [Ch. 9]

serializable—A property of before-or-after 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. [Ch. 9]

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 while 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. [Ch. 9]

simple locking—A locking protocol for creating before-or-after actions requiring that no data be read or written before reaching the lock point. For the atomic action to also be all-or-nothing, a further requirement is that no locks be released before commit (or abort). Compare with *two-phase locking*. [Ch. 9]

simple serialization—An atomicity protocol requiring that each newly created atomic action must wait to begin execution until all previously started atomic actions are no longer pending. [Ch. 9]

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 before-or-after. [Ch. 9]

transactional memory—A memory model in which multiple references to primary memory are both all-or-nothing and before-or-after. [Ch. 9]

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. [Ch. 9]

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*. [Ch. 9]

two-phase locking—A locking protocol for before-or-after atomicity 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*. [Ch. 9]

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*. [Ch. 9]

version history—The set of all values for an object or variable that have ever existed, stored in journal storage. [Ch. 9]

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. [Ch. 9]

write tearing—See *atomic storage*.

Index of Chapter 9

Design principles and hints appear in *underlined italics*. Procedure names appear in SMALL CAPS. Page numbers in **bold face** are in the chapter Glossary.

Principles of Computer System Design

An Introduction

Chapter 8
Fault Tolerance: Reliable Systems from Unreliable Components

Jerome H. Saltzer

M. Frans Kaashoek

Massachusetts Institute of Technology

Version 5.0

Suggestions, Comments, Corrections, and Requests to waive license restrictions: Please send correspondence by electronic mail to:

Saltzer@mit.edu

and

kaashoek@mit.edu

CHAPTER

Fault Tolerance: Reliable Systems from Unreliable Components

8

CHAPTER CONTENTS

Overview

Construction of reliable systems from unreliable components is one of the most important applications of modularity. There are, in principle, three basic steps to building reliable systems:

1. *Error detection*: discovering that there is an error in a data value or control signal. Error detection is accomplished with the help of *redundancy*, extra information that can verify correctness.

2. *Error containment*: limiting how far the effects of an error propagate. Error containment comes from careful application of modularity. When discussing reliability, a *module* is usually taken to be the unit that fails independently of other such units. It is also usually the unit of repair and replacement.

3. *Error masking*: ensuring correct operation despite the error. Error masking is accomplished by providing enough additional redundancy that it is possible to discover correct, or at least acceptably close, values of the erroneous data or control signal. When masking involves changing incorrect values to correct ones, it is usually called *error correction*.

Since these three steps can overlap in practice, one sometimes finds a single error-handling mechanism that merges two or even all three of the steps.

In earlier chapters each of these ideas has already appeared in specialized forms:

- A primary purpose of enforced modularity, as provided by client/server architecture, virtual memory, and threads, is error containment.

- Network links typically use error detection to identify and discard damaged frames.
- Some end-to-end protocols time out and resend lost data segments, thus masking the loss.
- Routing algorithms find their way around links that fail, masking those failures.
- Some real-time applications fill in missing data by interpolation or repetition, thus masking loss.

and, as we will see in Chapter 11[on-line], secure systems use a technique called *defense in depth* both to contain and to mask errors in individual protection mechanisms. In this chapter we explore systematic application of these techniques to more general problems, as well as learn about both their power and their limitations.

8.1 Faults, Failures, and Fault Tolerant Design

8.1.1 Faults, Failures, and Modules

Before getting into the techniques of constructing reliable systems, let us distinguish between concepts and give them separate labels. In ordinary English discourse, the three words "fault," "failure," and "error" are used more or less interchangeably or at least with strongly overlapping meanings. In discussing reliable systems, we assign these terms to distinct formal concepts. The distinction involves modularity. Although common English usage occasionally intrudes, the distinctions are worth maintaining in technical settings.

A *fault* is an underlying defect, imperfection, or flaw that has the potential to cause problems, whether it actually has, has not, or ever will. A weak area in the casing of a tire is an example of a fault. Even though the casing has not actually cracked yet, the fault is lurking. If the casing cracks, the tire blows out, and the car careens off a cliff, the resulting crash is a *failure*. (That definition of the term "failure" by example is too informal; we will give a more careful definition in a moment.) One fault that underlies the failure is the weak spot in the tire casing. Other faults, such as an inattentive driver and lack of a guard rail, may also contribute to the failure.

Experience suggests that faults are commonplace in computer systems. Faults come from many different sources: software, hardware, design, implementation, operations, and the environment of the system. Here are some typical examples:

- Software fault: A programming mistake, such as placing a less-than sign where there should be a less-than-or-equal sign. This fault may never have caused any trouble because the combination of events that requires the equality case to be handled correctly has not yet occurred. Or, perhaps it is the reason that the system crashes twice a day. If so, those crashes are failures.

- Hardware fault: A gate whose output is stuck at the value ZERO. Until something depends on the gate correctly producing the output value ONE, nothing goes wrong. If you publish a paper with an incorrect sum that was calculated by this gate, a failure has occurred. Furthermore, the paper now contains a fault that may lead some reader to do something that causes a failure elsewhere.
- Design fault: A miscalculation that has led to installing too little memory in a telephone switch. It may be months or years until the first time that the presented load is great enough that the switch actually begins failing to accept calls that its specification says it should be able to handle.
- Implementation fault: Installing less memory than the design called for. In this case the failure may be identical to the one in the previous example of a design fault, but the fault itself is different.
- Operations fault: The operator responsible for running the weekly payroll ran the payroll program twice last Friday. Even though the operator shredded the extra checks, this fault has probably filled the payroll database with errors such as wrong values for year-to-date tax payments.
- Environment fault: Lightning strikes a power line, causing a voltage surge. The computer is still running, but a register that was being updated at that instant now has several bits in error. Environment faults come in all sizes, from bacteria contaminating ink-jet printer cartridges to a storm surge washing an entire building out to sea.

Some of these examples suggest that a fault may either be *latent*, meaning that it isn't affecting anything right now, or *active*. When a fault is active, wrong results appear in data values or control signals. These wrong results are *errors*. If one has a formal specification for the design of a module, an error would show up as a violation of some assertion or invariant of the specification. The violation means that either the formal specification is wrong (for example, someone didn't articulate all of the assumptions) or a module that this component depends on did not meet its own specification. Unfortunately, formal specifications are rare in practice, so discovery of errors is more likely to be somewhat *ad hoc*.

If an error is not detected and masked, the module probably does not perform to its specification. Not producing the intended result at an interface is the formal definition of a *failure*. Thus, the distinction between fault and failure is closely tied to modularity and the building of systems out of well-defined subsystems. In a system built of subsystems, the failure of a subsystem is a fault from the point of view of the larger subsystem that contains it. That fault may cause an error that leads to the failure of the larger subsystem, unless the larger subsystem anticipates the possibility of the first one failing, detects the resulting error, and masks it. Thus, if you notice that you have a flat tire, you have detected an error caused by failure of a subsystem you depend on. If you miss an appointment because of the flat tire, the person you intended to meet notices a failure of a larger subsystem. If you change to a spare tire in time to get to the appointment, you have masked the error within your subsystem. *Fault tolerance* thus consists of noticing active faults and component subsystem failures and doing something helpful in response.

One such helpful response is *error containment*, which is another close relative of modularity and the building of systems out of subsystems. When an active fault causes an error in a subsystem, it may be difficult to confine the effects of that error to just a portion of the subsystem. On the other hand, one should expect that, as seen from outside that subsystem, the only effects will be at the specified interfaces of the subsystem. In consequence, the boundary adopted for error containment is usually the boundary of the smallest subsystem inside which the error occurred. From the point of view of the next higher-level subsystem, the subsystem with the error may contain the error in one of four ways:

1. Mask the error, so the higher-level subsystem does not realize that anything went wrong. One can think of failure as falling off a cliff and masking as a way of providing some separation from the edge.
2. Detect and report the error at its interface, producing what is called a *fail-fast* design. Fail-fast subsystems simplify the job of detection and masking for the next higher-level subsystem. If a fail-fast module correctly reports that its output is questionable, it has actually met its specification, so it has not failed. (Fail-fast modules can still fail, for example by not noticing their own errors.)
3. Immediately stop dead, thereby hoping to limit propagation of bad values, a technique known as *fail-stop*. Fail-stop subsystems require that the higher-level subsystem take some additional measure to discover the failure, for example by setting a timer and responding to its expiration. A problem with fail-stop design is that it can be difficult to distinguish a stopped subsystem from one that is merely running more slowly than expected. This problem is particularly acute in asynchronous systems.
4. Do nothing, simply failing without warning. At the interface, the error may have contaminated any or all output values. (Informally called a "crash" or perhaps "fail-thud".)

Another useful distinction is that of transient versus persistent faults. A *transient* fault, also known as a *single-event upset*, is temporary, triggered by some passing external event such as lightning striking a power line or a cosmic ray passing through a chip. It is usually possible to mask an error caused by a transient fault by trying the operation again. An error that is successfully masked by retry is known as a *soft error*. A *persistent* fault continues to produce errors, no matter how many times one retries, and the corresponding errors are called *hard errors*. An *intermittent* fault is a persistent fault that is active only occasionally, for example, when the noise level is higher than usual but still within specifications. Finally, it is sometimes useful to talk about *latency*, which in reliability terminology is the time between when a fault causes an error and when the error is detected or causes the module to fail. Latency can be an important parameter because some error-detection and error-masking mechanisms depend on there being at most a small fixed number of errors—often just one—at a time. If the error latency is large, there may be time for a second error to occur before the first one is detected and masked, in which case masking of the first error may not succeed. Also, a large error latency gives time for the error to propagate and may thus complicate containment.

Using this terminology, an improperly fabricated stuck-at-ZERO bit in a memory chip is a persistent fault: whenever the bit should contain a ONE the fault is active and the value of the bit is in error; at times when the bit is supposed to contain a ZERO, the fault is latent. If the chip is a component of a fault tolerant memory module, the module design probably includes an error-correction code that prevents that error from turning into a failure of the module. If a passing cosmic ray flips another bit in the same chip, a transient fault has caused that bit also to be in error, but the same error-correction code may still be able to prevent this error from turning into a module failure. On the other hand, if the error-correction code can handle only single-bit errors, the combination of the persistent and the transient fault might lead the module to produce wrong data across its interface, a failure of the module. If someone were then to test the module by storing new data in it and reading it back, the test would probably not reveal a failure because the transient fault does not affect the new data. Because simple input/output testing does not reveal successfully masked errors, a fault tolerant module design should always include some way to report that the module masked an error. If it does not, the user of the module may not realize that persistent errors are accumulating but hidden.

8.1.2 The Fault-Tolerance Design Process

One way to design a reliable system would be to build it entirely of components that are individually so reliable that their chance of failure can be neglected. This technique is known as *fault avoidance*. Unfortunately, it is hard to apply this technique to every component of a large system. In addition, the sheer number of components may defeat the strategy. If all N of the components of a system must work, the probability of any one component failing is p, and component failures are independent of one another, then the probability that the system works is $(1-p)^N$. No matter how small p may be, there is some value of N beyond which this probability becomes too small for the system to be useful.

The alternative is to apply various techniques that are known collectively by the name *fault tolerance*. The remainder of this chapter describes several such techniques that are the elements of an overall design process for building reliable systems from unreliable components. Here is an overview of the *fault-tolerance design process*:

1. Begin to develop a fault-tolerance model, as described in Section 8.3:

- Identify every potential fault.
- Estimate the risk of each fault, as described in Section 8.2.
- Where the risk is too high, design methods to detect the resulting errors.

2. Apply modularity to contain the damage from the high-risk errors.
3. Design and implement procedures that can mask the detected errors, using the techniques described in Section 8.4:

- Temporal redundancy. Retry the operation, using the same components.
- Spatial redundancy. Have different components do the operation.

4. Update the fault-tolerance model to account for those improvements.
5. Iterate the design and the model until the probability of untolerated faults is low enough that it is acceptable.
6. Observe the system in the field:

- Check logs of how many errors the system is successfully masking. (Always keep track of the distance to the edge of the cliff.)
- Perform postmortems on failures and identify *all* of the reasons for each failure.

7. Use the logs of masked faults and the postmortem reports about failures to revise and improve the fault-tolerance model and reiterate the design.

The fault-tolerance design process includes some subjective steps, for example, deciding that a risk of failure is "unacceptably high" or that the "probability of an untolerated fault is low enough that it is acceptable." It is at these points that different application requirements can lead to radically different approaches to achieving reliability. A personal computer may be designed with no redundant components, the computer system for a small business is likely to make periodic backup copies of most of its data and store the backup copies at another site, and some space-flight guidance systems use five completely redundant computers designed by at least two independent vendors. The decisions required involve trade-offs between the cost of failure and the cost of implementing fault tolerance. These decisions can blend into decisions involving business models and risk management. In some cases it may be appropriate to opt for a nontechnical solution, for example, deliberately accepting an increased risk of failure and covering that risk with insurance.

The fault-tolerance design process can be described as a *safety-net* approach to system design. The safety-net approach involves application of some familiar design principles and also some not previously encountered. It starts with a new design principle:

Be explicit

Get all of the assumptions out on the table.

The primary purpose of creating a fault-tolerance model is to expose and document the assumptions and articulate them explicitly. The designer needs to have these assumptions not only for the initial design, but also in order to respond to field reports of unexpected failures. Unexpected failures represent omissions or violations of the assumptions.

Assuming that you won't get it right the first time, the second design principle of the safety-net approach is the familiar *design for iteration.* It is difficult or impossible to anticipate all of the ways that things can go wrong. Moreover, when working with a fast-changing technology it can be hard to estimate probabilities of failure in components and in their organization, especially when the organization is controlled by software. For these reasons, a fault tolerant design must include feedback about actual error rates, evaluation of that feedback, and update of the design as field experience is gained. These two principles interact: to act on the feedback requires having a fault tolerance model that is explicit about reliability assumptions.

The third design principle of the safety-net approach is also familiar: the *safety margin principle*, described near the end of Section 1.3.2. An essential part of a fault tolerant design is to monitor how often errors are masked. When fault tolerant systems fail, it is usually not because they had inadequate fault tolerance, but because the number of failures grew unnoticed until the fault tolerance of the design was exceeded. The key requirement is that the system log all failures and that someone pay attention to the logs. The biggest difficulty to overcome in applying this principle is that it is hard to motivate people to expend effort checking something that seems to be working.

The fourth design principle of the safety-net approach came up in the introduction to the study of systems; it shows up here in the instruction to identify all of the causes of each failure: *keep digging*. Complex systems fail for complex reasons. When a failure of a system that is supposed to be reliable does occur, always look beyond the first, obvious cause. It is nearly always the case that there are actually several contributing causes and that there was something about the mind set of the designer that allowed each of those causes to creep in to the design.

Finally, complexity increases the chances of mistakes, so it is an enemy of reliability. The fifth design principle embodied in the safety-net approach is to *adopt sweeping simplifications*. This principle does not show up explicitly in the description of the fault-tolerance design process, but it will appear several times as we go into more detail.

The safety-net approach is applicable not just to fault tolerant design. Chapter 11[on-line] will show that the safety-net approach is used in an even more rigorous form in designing systems that must protect information from malicious actions.

8.2 Measures of Reliability and Failure Tolerance

8.2.1 Availability and Mean Time to Failure

A useful model of a system or a system component, from a reliability point of view, is that it operates correctly for some period of time and then it fails. The time to failure (TTF) is thus a measure of interest, and it is something that we would like to be able to predict. If a higher-level module does not mask the failure and the failure is persistent, the system cannot be used until it is repaired, perhaps by replacing the failed component, so we are equally interested in the time to repair (TTR). If we observe a system through N run–fail–repair cycles and observe in each cycle i the values of TTF_i and TTR_i, we can calculate the fraction of time it operated properly, a useful measure known as *availability:*

$$Availability = \frac{\text{time system was running}}{\text{time system should have been running}}$$

$$= \frac{\sum_{i=1}^{N} TTF_i}{\sum_{i=1}^{N} (TTF_i + TTR_i)} \qquad \text{Eq. 8–1}$$

By separating the denominator of the availability expression into two sums and dividing each by N (the number of observed failures) we obtain two time averages that are frequently reported as operational statistics: the *mean time to failure* (MTTF) and the *mean time to repair* (MTTR):

$$MTTF = \frac{1}{N}\sum_{i=1}^{N} TTF_i \qquad MTTR = \frac{1}{N}\sum_{i=1}^{N} TTR_i \qquad \text{Eq. 8–2}$$

The sum of these two statistics is usually called the *mean time between failures* (MTBF). Thus availability can be variously described as

$$Availability = \frac{MTTF}{MTBF} = \frac{MTTF}{MTTF + MTTR} = \frac{MTBF - MTTR}{MTBF} \qquad \text{Eq. 8–3}$$

In some situations, it is more useful to measure the fraction of time that the system is not working, known as its *down time*:

$$\text{Down time} = (1 - Availability) = \frac{MTTR}{MTBF} \qquad \text{Eq. 8–4}$$

One thing that the definition of down time makes clear is that MTTR and MTBF are in some sense equally important. One can reduce down time either by reducing MTTR or by increasing MTBF.

Components are often repaired by simply replacing them with new ones. When failed components are discarded rather than fixed and returned to service, it is common to use a slightly different method to measure MTTF. The method is to place a batch of N components in service in different systems (or in what is hoped to be an equivalent test environment), run them until they have all failed, and use the set of failure times as the TTF_i in equation 8–2. This procedure substitutes an ensemble average for the time average. We could use this same procedure on components that are not usually discarded when they fail, in the hope of determining their MTTF more quickly, but we might obtain a different value for the MTTF. Some failure processes do have the property that the ensemble average is the same as the time average (processes with this property are called *ergodic*), but other failure processes do not. For example, the repair itself may cause wear, tear, and disruption to other parts of the system, in which case each successive system failure might on average occur sooner than did the previous one. If that is the case, an MTTF calculated from an ensemble-average measurement might be too optimistic.

As we have defined them, availability, MTTF, MTTR, and MTBF are backward-looking measures. They are used for two distinct purposes: (1) for evaluating how the system is doing (compared, for example, with predictions made when the system was designed) and (2) for predicting how the system will behave in the future. The first purpose is concrete and well defined. The second requires that one take on faith that samples from the past provide an adequate predictor of the future, which can be a risky assumption. There are other problems associated with these measures. While MTTR can usually be measured in the field, the more reliable a component or system the longer it takes to evaluate its MTTF, so that measure is often not directly available. Instead, it is common to use and measure proxies to estimate its value. The quality of the resulting estimate of availability then depends on the quality of the proxy.

A typical 3.5-inch magnetic disk comes with a reliability specification of 300,000 hours "MTTF", which is about 34 years. Since the company quoting this number has probably not been in business that long, it is apparent that whatever they are calling "MTTF" is not the same as either the time-average or the ensemble-average MTTF that we just defined. It is actually a quite different statistic, which is why we put quotes around its name. Sometimes this "MTTF" is a theoretical prediction obtained by modeling the ways that the components of the disk might be expected to fail and calculating an expected time to failure.

A more likely possibility is that the manufacturer measured this "MTTF" by running an array of disks simultaneously for a much shorter time and counting the number of failures. For example, suppose the manufacturer ran 1,000 disks for 3,000 hours (about four months) each, and during that time 10 of the disks failed. The observed failure rate of this sample is 1 failure for every 300,000 hours of operation. The next step is to invert the failure rate to obtain 300,000 hours of operation per failure and then quote this number as the "MTTF". But the relation between this sample observation of failure rate and the real MTTF is problematic. If the failure process were memoryless (meaning that the failure rate is independent of time; Section 8.2.2, below, explores this idea more thoroughly), we would have the special case in which the MTTF really is the inverse of the failure rate. A good clue that the disk failure process is not memoryless is that the disk specification may also mention an "expected operational lifetime" of only 5 years. That statistic is probably the real MTTF—though even that may be a prediction based on modeling rather than a measured ensemble average. An appropriate re-interpretation of the 34-year "MTTF" statistic is to invert it and identify the result as a *short-term* failure rate that applies only within the expected operational lifetime. The paragraph discussing equation 8–9 on page 8–13 describes a fallacy that sometimes leads to miscalculation of statistics such as the MTTF.

Magnetic disks, light bulbs, and many other components exhibit a time-varying statistical failure rate known as a *bathtub curve*, illustrated in Figure 8.1 and defined more carefully in Section 8.2.2, below. When components come off the production line, a certain fraction fail almost immediately because of gross manufacturing defects. Those components that survive this initial period usually run for a long time with a relatively uniform failure rate. Eventually, accumulated wear and tear cause the failure rate to increase again, often quite rapidly, producing a failure rate plot that resembles the shape of a bathtub.

Several other suggestive and colorful terms describe these phenomena. Components that fail early are said to be subject to *infant mortality,* and those that fail near the end of their expected lifetimes are said to *burn out*. Manufacturers sometimes *burn in* such components by running them for a while before shipping, with the intent of identifying and discarding the ones that would otherwise fail immediately upon being placed in service. When a vendor quotes an "expected operational lifetime," it is probably the mean time to failure of those components that survive burn in, while the much larger "MTTF" number is probably the inverse of the observed failure rate at the lowest point of the bathtub. (The published numbers also sometimes depend on the outcome of a debate between the legal department and the marketing department, but that gets us into a different topic.) A chip manufacturer describes the fraction of components that survive the burn-in period as the *yield* of the production line. Component manufacturers usually exhibit a phenomenon known informally as a *learning curve,* which simply means that the first components coming out of a new production line tend to have more failures than later ones. The reason is that manufacturers *design for iteration*: upon seeing and analyzing failures in the early production batches, the production line designer figures out how to refine the manufacturing process to reduce the infant mortality rate.

One job of the system designer is to exploit the nonuniform failure rates predicted by the bathtub and learning curves. For example, a conservative designer exploits the learning curve by avoiding the latest generation of hard disks in favor of slightly older designs that have accumulated more field experience. One can usually rely on other designers who may be concerned more about cost or performance than availability to shake out the bugs in the newest generation of disks.

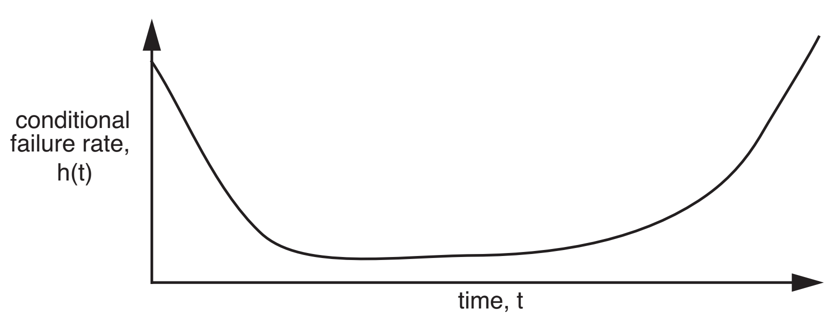


FIGURE 8.1

A bathtub curve, showing how the conditional failure rate of a component changes with time.

The 34-year "MTTF" disk drive specification may seem like public relations puffery in the face of the specification of a 5-year expected operational lifetime, but these two numbers actually are useful as a measure of the nonuniformity of the failure rate. This nonuniformity is also susceptible to exploitation, depending on the operation plan. If the operation plan puts the component in a system such as a satellite, in which it will run until it fails, the designer would base system availability and reliability estimates on the 5-year figure. On the other hand, the designer of a ground-based storage system, mindful that the 5-year operational lifetime identifies the point where the conditional failure rate starts to climb rapidly at the far end of the bathtub curve, might include a plan to replace perfectly good hard disks before burn-out begins to dominate the failure rate—in this case, perhaps every 3 years. Since one can arrange to do scheduled replacement at convenient times, for example, when the system is down for another reason, or perhaps even without bringing the system down, the designer can minimize the effect on system availability. The manufacturer's 34-year "MTTF", which is probably the inverse of the observed failure rate at the lowest point of the bathtub curve, then can be used as an estimate of the expected rate of unplanned replacements, although experience suggests that this specification may be a bit optimistic. Scheduled replacements are an example of *preventive maintenance*, which is active intervention intended to increase the mean time to failure of a module or system and thus improve availability.

For some components, observed failure rates are so low that MTTF is estimated by *accelerated aging*. This technique involves making an educated guess about what the dominant underlying cause of failure will be and then amplifying that cause. For example, it is conjectured that failures in recordable Compact Disks are heat-related. A typical test scenario is to store batches of recorded CDs at various elevated temperatures for several months, periodically bringing them out to test them and count how many have failed. One then plots these failure rates versus temperature and extrapolates to estimate what the failure rate would have been at room temperature. Again making the assumption that the failure process is memoryless, that failure rate is then inverted to produce an MTTF. Published MTTFs of 100 years or more have been obtained this way. If the dominant fault mechanism turns out to be something else (such as bacteria munching on the plastic coating) or if after 50 years the failure process turns out not to be memoryless after all, an estimate from an accelerated aging study may be far wide of the mark. A designer must use such estimates with caution and understanding of the assumptions that went into them.

Availability is sometimes discussed by counting the number of nines in the numerical representation of the availability measure. Thus a system that is up and running 99.9% of the time is said to have 3-nines availability. Measuring by nines is often used in marketing because it sounds impressive. A more meaningful number is usually obtained by calculating the corresponding down time. A 3-nines system can be down nearly 1.5 minutes per day or 8 hours per year, a 5-nines system 5 minutes per year, and a 7-nines system only 3 seconds per year. Another problem with measuring by nines is that it tells only about availability, without any information about MTTF. One 3-nines system may have a brief failure every day, while a different 3-nines system may have a single eight hour outage once a year. Depending on the application, the difference between those two systems could be important. Any single measure should always be suspect.

Finally, availability can be a more fine-grained concept. Some systems are designed so that when they fail, some functions (for example, the ability to read data) remain available, while others (the ability to make changes to the data) are not. Systems that continue to provide partial service in the face of failure are called *fail-soft*, a concept defined more carefully in Section 8.3.

8.2.2 Reliability Functions

The bathtub curve expresses the conditional failure rate *h(t)* of a module, defined to be the probability that the module fails between time *t* and time *t + dt,* given that the component is still working at time *t.* The conditional failure rate is only one of several closely related ways of describing the failure characteristics of a component, module, or system. The *reliability*, *R,* of a module is defined to be

$$R(t) = Pr\left(\begin{array}{c}\text{the module has not yet failed at time t, given that} \\ \text{the module was operating at time 0}\end{array}\right) \qquad \text{Eq. 8–5}$$

and the unconditional failure rate *f(t)* is defined to be

$$f(t) = Pr(\text{module fails between t and t + dt}) \qquad \text{Eq. 8–6}$$

(The bathtub curve and these two reliability functions are three ways of presenting the same information. If you are rusty on probability, a brief reminder of how they are related appears in Sidebar 8.1.) Once *f(t)* is at hand, one can directly calculate the MTTF:

$$MTTF = \int_0^{\infty} t \cdot f(t)dt \qquad \text{Eq. 8–7}$$

One must keep in mind that this MTTF is predicted from the failure rate function *f(t)*, in contrast to the MTTF of eq. 8–2, which is the result of a field measurement. The two MTTFs will be the same only if the failure model embodied in *f(t)* is accurate.

Some components exhibit relatively uniform failure rates, at least for the lifetime of the system of which they are a part. For these components the conditional failure rate, rather than resembling a bathtub, is a straight horizontal line, and the reliability function becomes a simple declining exponential:

$$R(t) = e^{-\left(\frac{t}{MTTF}\right)} \qquad \text{Eq. 8–8}$$

This reliability function is said to be *memoryless*, which simply means that the conditional failure rate is independent of how long the component has been operating. Memoryless failure processes have the nice property that the conditional failure rate is the inverse of the MTTF.

Unfortunately, as we saw in the case of the disks with the 34-year "MTTF", this property is sometimes misappropriated to quote an MTTF for a component whose

Sidebar 8.1: Reliability functions The failure rate function, the reliability function, and the bathtub curve (which in probability texts is called the *conditional failure rate function*, and which in operations research texts is called the *hazard function*) are actually three mathematically related ways of describing the same information. The failure rate function, $f(t)$ as defined in equation 8–6, is a *probability density function*, which is everywhere non-negative and whose integral over all time is 1. Integrating the failure rate function from the time the component was created (conventionally taken to be $t = 0$) to the present time yields

$$F(t) = \int_0^t f(t)dt$$

$F(t)$ is the cumulative probability that the component has failed by time t. The cumulative probability that the component has *not* failed is the probability that it is still operating at time t given that it was operating at time 0, which is exactly the definition of the reliability function, $R(t)$. That is,

$$R(t) = 1 - F(t)$$

The bathtub curve of Figure 8.1 reports the conditional probability $h(t)$ that a failure occurs between t and t + dt, given that the component was operating at time t. By the definition of conditional probability, the conditional failure rate function is thus

$$h(t) = \frac{f(t)}{R(t)}$$

conditional failure rate does change with time. This misappropriation starts with a fallacy: an assumption that the MTTF, as defined in eq. 8–7, can be calculated by inverting the measured failure rate. The fallacy arises because in general,

$$E(1/t) \neq 1/E(t) \qquad \text{Eq. 8–9}$$

That is, the expected value of the inverse is *not* equal to the inverse of the expected value, except in certain special cases. The important special case in which they *are* equal is the memoryless distribution of eq. 8–8. When a random process is memoryless, calculations and measurements are so much simpler that designers sometimes forget that the same simplicity does not apply everywhere.

Just as availability is sometimes expressed in an oversimplified way by counting the number of nines in its numerical representation, reliability in component manufacturing is sometimes expressed in an oversimplified way by counting standard deviations in the observed distribution of some component parameter, such as the maximum propagation time of a gate. The usual symbol for standard deviation is the Greek letter σ (sigma), and a normal distribution has a standard deviation of 1.0, so saying that a component has "4.5 σ reliability" is a shorthand way of saying that the production line controls variations in that parameter well enough that the specified tolerance is 4.5 standard deviations away from the mean value, as illustrated in Figure 8.2. Suppose, for example, that a production line is manufacturing gates that are specified to have a mean propagation time of 10 nanoseconds and a maximum propagation time of 11.8 nanoseconds with 4.5 σ reliability. The difference between the mean and the maximum, 1.8 nanoseconds, is the tolerance. For that tolerance to be 4.5 σ, σ would have to be no more than 0.4 nanoseconds. To meet the specification, the production line designer would measure the actual propagation times of production line samples and, if the observed variance is greater than 0.4 ns, look for ways to reduce the variance to that level.

Another way of interpreting "4.5 σ reliability" is to calculate the expected fraction of components that are outside the specified tolerance. That fraction is the integral of one tail of the normal distribution from 4.5 to ∞, which is about 3.4×10^{-6} , so in our example no more than 3.4 out of each million gates manufactured would have delays greater than 11.8 nanoseconds. Unfortunately, this measure describes only the failure rate of the production line, it does not say anything about the failure rate of the component after it is installed in a system.

A currently popular quality control method, known as "Six Sigma", is an application of two of our design principles to the manufacturing process. The idea is to use measurement, feedback, and iteration (*design for iteration*: "you won't get it right the first time") to reduce the variance (*the robustness principle*: "be strict on outputs") of production-line manufacturing. The "Six Sigma" label is somewhat misleading because in the application of the method, the number 6 is allocated to deal with two quite different effects. The method sets a target of controlling the production line variance to the level of 4.5 σ, just as in the gate example of Figure 8.2. The remaining 1.5 σ is the amount that the mean output value is allowed to drift away from its original specification over the life of the

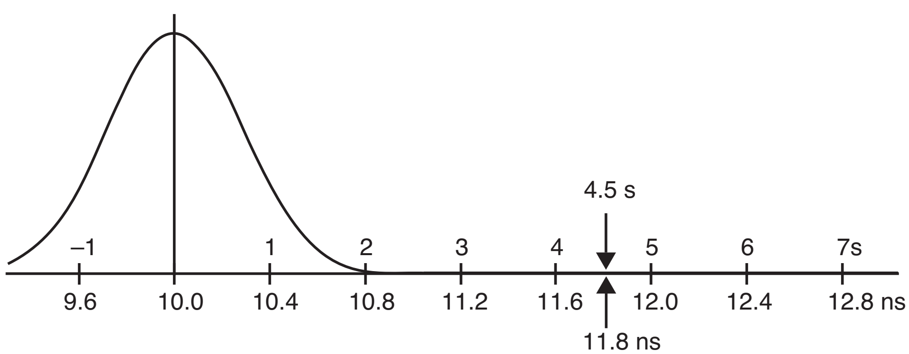


FIGURE 8.2

The normal probability density function applied to production of gates that are specified to have mean propagation time of 10 nanoseconds and maximum propagation time of 11.8 nanoseconds. The upper numbers on the horizontal axis measure the distance from the mean in units of the standard deviation, σ. The lower numbers depict the corresponding propagation times. The integral of the tail from 4.5 σ to ∞ is so small that it is not visible in this figure.

production line. So even though the production line may start 6 σ away from the tolerance limit, after it has been operating for a while one may find that the failure rate has drifted upward to the same 3.4 in a million calculated for the 4.5 σ case.

In manufacturing quality control literature, these applications of the two design principles are known as *Taguchi methods*, after their popularizer, Genichi Taguchi.

8.2.3 Measuring Fault Tolerance

It is sometimes useful to have a quantitative measure of the fault tolerance of a system. One common measure, sometimes called the *failure tolerance*, is the number of failures of its components that a system can tolerate without itself failing. Although this label could be ambiguous, it is usually clear from context that a measure is being discussed. Thus a memory system that includes single-error correction (Section 8.4 describes how error correction works) has a failure tolerance of one bit.

When a failure occurs, the remaining failure tolerance of the system goes down. The remaining failure tolerance is an important thing to monitor during operation of the system because it shows how close the system as a whole is to failure. One of the most common system design mistakes is to add fault tolerance but not include any monitoring to see how much of the fault tolerance has been used up, thus ignoring the *safety margin principle*. When systems that are nominally fault tolerant do fail, later analysis invariably discloses that there were several failures that the system successfully masked but that somehow were never reported and thus were never repaired. Eventually, the total number of failures exceeded the designed failure tolerance of the system.

Failure tolerance is actually a single number in only the simplest situations. Sometimes it is better described as a vector, or even as a matrix showing the specific combinations of different kinds of failures that the system is designed to tolerate. For example, an electric power company might say that it can tolerate the failure of up to 15% of its generating capacity, at the same time as the downing of up to two of its main transmission lines.

8.3 Tolerating Active Faults

8.3.1 Responding to Active Faults

In dealing with active faults, the designer of a module can provide one of several responses:

- *Do nothing.* The error becomes a failure of the module, and the larger system or subsystem of which it is a component inherits the responsibilities both of discovering and of handling the problem. The designer of the larger subsystem then must choose which of these responses to provide. In a system with several layers of modules, failures may be passed up through more than one layer before being discovered and handled. As the number of do-nothing layers increases, containment generally becomes more and more difficult.

- *Be fail-fast.* The module reports at its interface that something has gone wrong. This response also turns the problem over to the designer of the next higher-level system, but in a more graceful way. Example: when an Ethernet transceiver detects a collision on a frame it is sending, it stops sending as quickly as possible, broadcasts a brief jamming signal to ensure that all network participants quickly realize that there was a collision, and it reports the collision to the next higher level, usually a hardware module of which the transceiver is a component, so that the higher level can consider resending that frame.
- *Be fail-safe.* The module transforms any value or values that are incorrect to values that are known to be acceptable, even if not right or optimal. An example is a digital traffic light controller that, when it detects a failure in its sequencer, switches to a blinking red light in all directions. Chapter 11[on-line] discusses systems that provide security. In the event of a failure in a secure system, the safest thing to do is usually to block all access. A fail-safe module designed to do that is said to be *fail-secure*.
- *Be fail-soft.* The system continues to operate correctly with respect to some predictably degraded subset of its specifications, perhaps with some features missing or with lower performance. For example, an airplane with three engines can continue to fly safely, albeit more slowly and with less maneuverability, if one engine fails. A file system that is partitioned into five parts, stored on five different small hard disks, can continue to provide access to 80% of the data when one of the disks fails, in contrast to a file system that employs a single disk five times as large.
- *Mask the error.* Any value or values that are incorrect are made right and the module meets it specification as if the error had not occurred.

We will concentrate on masking errors because the techniques used for that purpose can be applied, often in simpler form, to achieving a fail-fast, fail-safe, or fail-soft system.

As a general rule, one can design algorithms and procedures to cope only with specific, anticipated faults. Further, an algorithm or procedure can be expected to cope only with faults that are actually detected. In most cases, the only workable way to detect a fault is by noticing an incorrect value or control signal; that is, by detecting an error. Thus when trying to determine if a system design has adequate fault tolerance, it is helpful to classify errors as follows:

- A *detectable error* is one that can be detected reliably. If a detection procedure is in place and the error occurs, the system discovers it with near certainty and it becomes a *detected* error.

- A *maskable error* is one for which it is possible to devise a procedure to recover correctness. If a masking procedure is in place and the error occurs, is detected, and is masked, the error is said to be *tolerated*.
- Conversely, an *untolerated* error is one that is undetectable, undetected, unmaskable, or unmasked.

An untolerated error usually leads to a failure of the system. ("Usually," because we could get lucky and still produce a correct output, either because the error values didn't actually matter under the current conditions, or some measure intended to mask a different error incidentally masks this one, too.) This classification of errors is illustrated in Figure 8.3.

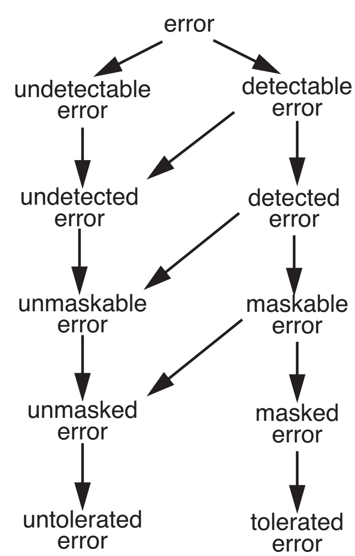


FIGURE 8.3

Classification of errors. Arrows lead from a category to mutually exclusive subcategories. For example, unmasked errors include both unmaskable errors and maskable errors that the designer decides not to mask.

A subtle consequence of the concept of a maskable error is that there must be a well-defined boundary around that part of the system state that might be in error. The masking procedure must restore all of that erroneous state to correctness, using information that has not been corrupted by the error. The real meaning of detectable, then, is that the error is discovered before its consequences have propagated beyond some specified boundary. The designer usually chooses this boundary to coincide with that of some module and designs that module to be fail-fast (that is, it detects and reports its own errors). The system of which the module is a component then becomes responsible for masking the failure of the module.

8.3.2 Fault Tolerance Models

The distinctions among detectable, detected, maskable, and tolerated errors allow us to specify for a system a *fault tolerance model*, one of the components of the fault tolerance design process described in Section 8.1.2, as follows:

1. Analyze the system and categorize possible error events into those that can be reliably detected and those that cannot. At this stage, detectable or not, all errors are untolerated.

2. For each undetectable error, evaluate the probability of its occurrence. If that probability is not negligible, modify the system design in whatever way necessary to make the error reliably detectable.
3. For each detectable error, implement a detection procedure and reclassify the module in which it is detected as fail-fast.
4. For each detectable error try to devise a way of masking it. If there is a way, reclassify this error as a maskable error.
5. For each maskable error, evaluate its probability of occurrence, the cost of failure, and the cost of the masking method devised in the previous step. If the evaluation indicates it is worthwhile, implement the masking method and reclassify this error as a tolerated error.

When finished developing such a model, the designer should have a useful fault tolerance specification for the system. Some errors, which have negligible probability of occurrence or for which a masking measure would be too expensive, are identified as untolerated. When those errors occur the system fails, leaving its users to cope with the result. Other errors have specified recovery algorithms, and when those occur the system should continue to run correctly. A review of the system recovery strategy can now focus separately on two distinct questions:

- Is the designer's list of potential error events complete, and is the assessment of the probability of each error realistic?
- Is the designer's set of algorithms, procedures, and implementations that are supposed to detect and mask the anticipated errors complete and correct?

These two questions are different. The first is a question of models of the real world. It addresses an issue of experience and judgment about real-world probabilities and whether all real-world modes of failure have been discovered or some have gone unnoticed. Two different engineers, with different real-world experiences, may reasonably disagree on such judgments—they may have different models of the real world. The evaluation of modes of failure and of probabilities is a point at which a designer may easily go astray because such judgments must be based not on theory but on experience in the field, either personally acquired by the designer or learned from the experience of others. A new technology, or an old technology placed in a new environment, is likely to create surprises. A wrong judgment can lead to wasted effort devising detection and masking algorithms that will rarely be invoked rather than the ones that are really needed. On the other hand, if the needed experience is not available, all is not lost: the iteration part of the design process is explicitly intended to provide that experience.

The second question is more abstract and also more absolutely answerable, in that an argument for correctness (unless it is hopelessly complicated) or a counterexample to that argument should be something that everyone can agree on. In system design, it is helpful to follow design procedures that distinctly separate these classes of questions. When someone questions a reliability feature, the designer can first ask, "Are you questioning the correctness of my recovery algorithm or are you questioning my model of what may fail?" and thereby properly focus the discussion or argument.

Creating a fault tolerance model also lays the groundwork for the iteration part of the fault tolerance design process. If a system in the field begins to fail more often than expected, or completely unexpected failures occur, analysis of those failures can be compared with the fault tolerance model to discover what has gone wrong. By again asking the two questions marked with bullets above, the model allows the designer to distinguish between, on the one hand, failure probability predictions being proven wrong by field experience, and on the other, inadequate or misimplemented masking procedures. With this information the designer can work out appropriate adjustments to the model and the corresponding changes needed for the system.

Iteration and review of fault tolerance models is also important to keep them up to date in the light of technology changes. For example, the Network File System described in Section 4.4 was first deployed using a local area network, where packet loss errors are rare and may even be masked by the link layer. When later users deployed it on larger networks, where lost packets are more common, it became necessary to revise its fault tolerance model and add additional error detection in the form of end-to-end checksums. The processor time required to calculate and check those checksums caused some performance loss, which is why its designers did not originally include checksums. But loss of data integrity outweighed loss of performance and the designers reversed the trade-off.

To illustrate, an example of a fault tolerance model applied to a popular kind of memory devices, RAM, appears in Section 8.7. This fault tolerance model employs error detection and masking techniques that are described below in Section 8.4 of this chapter, so the reader may prefer to delay detailed study of that section until completing Section 8.4.

8.4 Systematically Applying Redundancy

The designer of an analog system typically masks small errors by specifying design tolerances known as *margins*, which are amounts by which the specification is better than necessary for correct operation under normal conditions. In contrast, the designer of a digital system both detects and masks errors of all kinds by adding redundancy, either in time or in space. When an error is thought to be transient, as when a packet is lost in a data communication network, one method of masking is to resend it, an example of redundancy in time. When an error is likely to be persistent, as in a failure in reading bits from the surface of a disk, the usual method of masking is with spatial redundancy, having another component provide another copy of the information or control signal. Redundancy can be applied either in cleverly small quantities or by brute force, and both techniques may be used in different parts of the same system.

8.4.1 Coding: Incremental Redundancy

The most common form of incremental redundancy, known as *forward error correction*, consists of clever coding of data values. With data that has not been encoded to tolerate errors, a change in the value of one bit may transform one legitimate data value into another legitimate data value. Encoding for errors involves choosing as the representation of legitimate data values only some of the total number of possible bit patterns, being careful that the patterns chosen for legitimate data values all have the property that to transform any one of them to any other, more than one bit must change. The smallest number of bits that must change to transform one legitimate pattern into another is known as the *Hamming distance* between those two patterns. The Hamming distance is named after Richard Hamming, who first investigated this class of codes. Thus the patterns

```
100101
000111
```

have a Hamming distance of 2 because the upper pattern can be transformed into the lower pattern by flipping the values of two bits, the first bit and the fifth bit. Data fields that have not been coded for errors might have a Hamming distance as small as 1. Codes that can detect or correct errors have a minimum Hamming distance between any two legitimate data patterns of 2 or more. The Hamming distance of a code is the minimum Hamming distance between any pair of legitimate patterns of the code. One can calculate the Hamming distance between two patterns, *A* and *B*, by counting the number of ONES in $A \oplus B$, where $\oplus$ is the exclusive OR (XOR) operator.

Suppose we create an encoding in which the Hamming distance between *every* pair of legitimate data patterns is 2. Then, if one bit changes accidentally, since no legitimate data item can have that pattern, we can detect that something went wrong, but it is not possible to figure out what the original data pattern was. Thus, if the two patterns above were two members of the code and the first bit of the upper pattern were flipped from ONE to ZERO, there is no way to tell that the result, `000101`, is not the result of flipping the fifth bit of the lower pattern.

Next, suppose that we instead create an encoding in which the Hamming distance of the code is 3 or more. Here are two patterns from such a code; bits 1, 2, and 5 are different:

```
100101
010111
```

Now, a one-bit change will always transform a legitimate data pattern into an incorrect data pattern that is still at least 2 bits distant from any other legitimate pattern but only 1 bit distant from the original pattern. A decoder that receives a pattern with a one-bit error can inspect the Hamming distances between the received pattern and nearby legitimate patterns and by choosing the nearest legitimate pattern correct the error. If 2 bits change, this error-correction procedure will still identify a corrected data value, but it will choose the wrong one. If we expect 2-bit errors to happen often, we could choose the code patterns so that the Hamming distance is 4, in which case the code can correct

1-bit errors and detect 2-bit errors. But a 3-bit error would look just like a 1-bit error in some other code pattern, so it would decode to a wrong value. More generally, if the Hamming distance of a code is *d*, a little analysis reveals that one can detect $d-1$ errors and correct $\lfloor (d-1)/2 \rfloor$ errors. The reason that this form of redundancy is named "forward" error correction is that the creator of the data performs the coding before storing or transmitting it, and anyone can later decode the data without appealing to the creator. (Chapter 7[on-line] described the technique of asking the sender of a lost frame, packet, or message to retransmit it. That technique goes by the name of *backward error correction.*)

The systematic construction of forward error-detection and error-correction codes is a large field of study, which we do not intend to explore. However, two specific examples of commonly encountered codes are worth examining.

The first example is a simple parity check on a 2-bit value, in which the parity bit is the XOR of the 2 data bits. The coded pattern is 3 bits long, so there are $2^3 = 8$ possible patterns for this 3-bit quantity, only 4 of which represent legitimate data. As illustrated in Figure 8.4, the 4 "correct" patterns have the property that changing any single bit transforms the word into one of the 4 illegal patterns. To transform the coded quantity into another legal pattern, at least 2 bits must change (in other words, the Hamming distance of this code is 2). The conclusion is that a simple parity check can detect any single error, but it doesn't have enough information to correct errors.

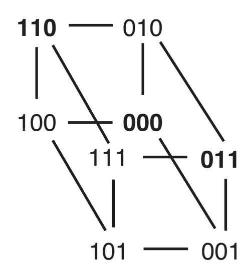


FIGURE 8.4

Patterns for a simple parity-check code. Each line connects patterns that differ in only one bit; **bold-face** patterns are the legitimate ones.

The second example, in Figure 8.5, shows a forward error-correction code that can correct 1-bit errors in a 4-bit data value, by encoding the 4 bits into 7-bit words. In this code, bits P_7, P_6, P_5, and P_3 carry the data, while bits P_4, P_2, and P_1 are calculated from the data bits. (This out-of-order numbering scheme creates a multidimensional binary coordinate system with a use that will be evident in a moment.) We could analyze this code to determine its Hamming distance, but we can also observe that three extra bits can carry exactly enough information to distinguish 8 cases: no error, an error in bit 1, an error in bit 2, … or an error in bit 7. Thus, it is not surprising that an error-correction code can be created. This code calculates bits P_1, P_2, and P_4 as follows:

$$P_1 = P_7 \oplus P_5 \oplus P_3$$
$$P_2 = P_7 \oplus P_6 \oplus P_3$$
$$P_4 = P_7 \oplus P_6 \oplus P_5$$

Now, suppose that the array of bits P_1 through P_7 is sent across a network and noise causes bit P_5 to flip. If the recipient recalculates P_1, P_2, and P_4, the recalculated values of P_1 and P_4 will be different from the received bits P_1 and P_4. The recipient then writes $P_4\ P_2\ P_1$ in order, representing the troubled bits as ONEs and untroubled bits as ZEROs, and notices that their binary value is $101_2 = 5$, the position of the flipped bit. In this code, whenever there is a one-bit error, the troubled parity bits directly identify the bit to correct. (That was the reason for the out-of-order bit-numbering scheme, which created a 3-dimensional coordinate system for locating an erroneous bit.)

The use of 3 check bits for 4 data bits suggests that an error-correction code may not be efficient, but in fact the apparent inefficiency of this example is only because it is so small. Extending the same reasoning, one can, for example, provide single-error correction for 56 data bits using 7 check bits in a 63-bit code word.

In both of these examples of coding, the assumed threat to integrity is that an unidentified bit out of a group may be in error. Forward error correction can also be effective against other threats. A different threat, called *erasure,* is also common in digital systems. An erasure occurs when the value of a particular, identified bit of a group is unintelligible or perhaps even completely missing. Since we know which bit is in question, the simple parity-check code, in which the parity bit is the XOR of the other bits, becomes a forward error correction code. The unavailable bit can be reconstructed simply by calculating the XOR of the unerased bits. Returning to the example of Figure 8.4, if we find a pattern in which the first and last bits have values 0 and 1 respectively, but the middle bit is illegible, the only possibilities are 001 and **011**. Since 001 is not a legitimate code pattern, the original pattern must have been **011**. The simple parity check allows correction of only a single erasure. If there is a threat of multiple erasures, a more complex coding scheme is needed. Suppose, for example, we have 4 bits to protect, and they are coded as in Figure 8.5. In that case, if as many as 3 bits are erased, the remaining 4 bits are sufficient to reconstruct the values of the 3 that are missing.

Since erasure, in the form of lost packets, is a threat in a best-effort packet network, this same scheme of forward error correction is applicable. One might, for example, send four numbered, identical-length packets of data followed by a parity packet that contains

	bit	P_7	P_6	P_5	P_4	P_3	P_2	P_1
Choose P_1 so XOR of every other bit ($P_7 \oplus P_5 \oplus P_3 \oplus P_1$) is 0		$\oplus$		$\oplus$		$\oplus$		$\oplus$
Choose P_2 so XOR of every other pair ($P_7 \oplus P_6 \oplus P_3 \oplus P_2$) is 0		$\oplus$	$\oplus$			$\oplus$	$\oplus$	
Choose P_4 so XOR of every other four ($P_7 \oplus P_6 \oplus P_5 \oplus P_4$) is 0		$\oplus$	$\oplus$	$\oplus$	$\oplus$			

FIGURE 8.5

A single-error-correction code. In the table, the symbol $\oplus$ marks the bits that participate in the calculation of one of the redundant bits. The payload bits are P_7, P_6, P_5, and P_3, and the redundant bits are P_4, P_2, and P_1. The "every other" notes describe a 3-dimensional coordinate system that can locate an erroneous bit.

as its payload the bit-by-bit XOR of the payloads of the previous four. (That is, the first bit of the parity packet is the XOR of the first bit of each of the other four packets; the second bits are treated similarly, etc.) Although the parity packet adds 25% to the network load, as long as any four of the five packets make it through, the receiving side can reconstruct all of the payload data perfectly without having to ask for a retransmission. If the network is so unreliable that more than one packet out of five typically gets lost, then one might send seven packets, of which four contain useful data and the remaining three are calculated using the formulas of Figure 8.5. (Using the numbering scheme of that figure, the payload of packet 4, for example, would consist of the XOR of the payloads of packets 7, 6, and 5.) Now, if any four of the seven packets make it through, the receiving end can reconstruct the data.

Forward error correction is especially useful in broadcast protocols, where the existence of a large number of recipients, each of which may miss different frames, packets, or stream segments, makes the alternative of backward error correction by requesting retransmission unattractive. Forward error correction is also useful when controlling jitter in stream transmission because it eliminates the round-trip delay that would be required in requesting retransmission of missing stream segments. Finally, forward error correction is usually the only way to control errors when communication is one-way or round-trip delays are so long that requesting retransmission is impractical, for example, when communicating with a deep-space probe. On the other hand, using forward error correction to replace lost packets may have the side effect of interfering with congestion control techniques in which an overloaded packet forwarder tries to signal the sender to slow down by discarding an occasional packet.

Another application of forward error correction to counter erasure is in storing data on magnetic disks. The threat in this case is that an entire disk drive may fail, for example because of a disk head crash. Assuming that the failure occurs long after the data was originally written, this example illustrates one-way communication in which backward error correction (asking the original writer to write the data again) is not usually an option. One response is to use a RAID array (see Section 2.1.1.4) in a configuration known as RAID 4. In this configuration, one might use an array of five disks, with four of the disks containing application data and each sector of the fifth disk containing the bit-by-bit XOR of the corresponding sectors of the first four. If any of the five disks fails, its identity will quickly be discovered because disks are usually designed to be fail-fast and report failures at their interface. After replacing the failed disk, one can restore its contents by reading the other four disks and calculating, sector by sector, the XOR of their data (see exercise 8.9). To maintain this strategy, whenever anyone updates a data sector, the RAID 4 system must also update the corresponding sector of the parity disk, as shown in Figure 8.6. That figure makes it apparent that, in RAID 4, forward error correction has an identifiable read and write performance cost, in addition to the obvious increase in the amount of disk space used. Since loss of data can be devastating, there is considerable interest in RAID, and much ingenuity has been devoted to devising ways of minimizing the performance penalty.

Although it is an important and widely used technique, successfully applying incremental redundancy to achieve error detection and correction is harder than one might expect. The first case study of Section 8.8 provides several useful lessons on this point.

In addition, there are some situations where incremental redundancy does not seem to be applicable. For example, there have been efforts to devise error-correction codes for numerical values with the property that the coding is preserved when the values are processed by an adder or a multiplier. While it is not too hard to invent schemes that allow a limited form of error detection (for example, one can verify that residues are consistent, using analogues of casting out nines, which school children use to check their arithmetic), these efforts have not yet led to any generally applicable techniques. The only scheme that has been found to systematically protect data during arithmetic processing is massive redundancy, which is our next topic.

8.4.2 Replication: Massive Redundancy

In designing a bridge or a skyscraper, a civil engineer masks uncertainties in the strength of materials and other parameters by specifying components that are 5 or 10 times as strong as minimally required. The method is heavy-handed, but simple and effective.

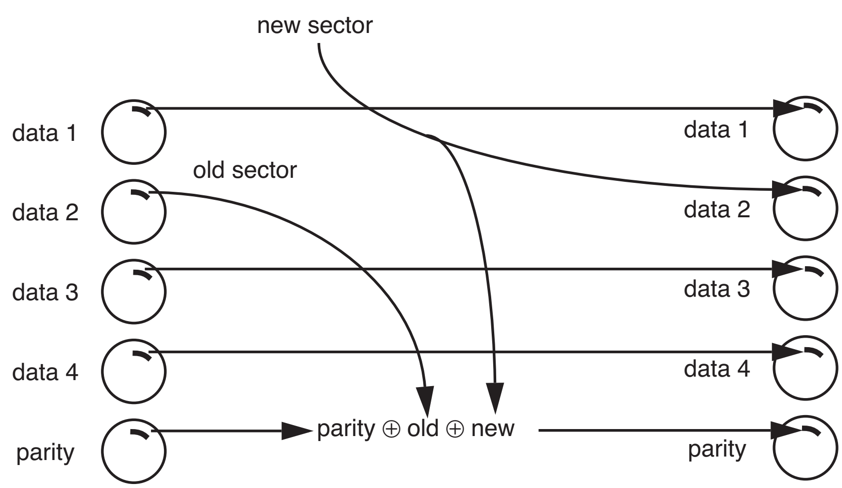


FIGURE 8.6

Update of a sector on disk 2 of a five-disk RAID 4 system. The old parity sector contains parity ← data 1 ⊕ data 2 ⊕ data 3 ⊕ data 4. To construct a new parity sector that includes the new data 2, one could read the corresponding sectors of data 1, data 3, and data 4 and perform three more XORS. But a faster way is to read just the old parity sector and the old data 2 sector and compute the new parity sector as

new parity ← old parity ⊕ old data 2 ⊕ new data 2

The corresponding way of building a reliable system out of unreliable discrete components is to acquire multiple copies of each component. Identical multiple copies are called *replicas*, and the technique is called *replication*. There is more to it than just making copies: one must also devise a plan to arrange or interconnect the replicas so that a failure in one replica is automatically masked with the help of the ones that don't fail. For example, if one is concerned about the possibility that a diode may fail by either shorting out or creating an open circuit, one can set up a network of four diodes as in Figure 8.7, creating what we might call a "superdiode". This interconnection scheme, known as a *quad component*, was developed by Claude E. Shannon and Edward F. Moore in the 1950s as a way of increasing the reliability of relays in telephone systems. It can also be used with resistors and capacitors in circuits that can tolerate a modest range of component values. This particular superdiode can tolerate a single short circuit *and* a single open circuit in any two component diodes, and it can also tolerate certain other multiple failures, such as open circuits in both upper diodes plus a short circuit in one of the lower diodes. If the bridging connection of the figure is added, the superdiode can tolerate additional multiple open-circuit failures (such as one upper diode and one lower diode), but it will be less tolerant of certain short-circuit failures (such as one left diode and one right diode).

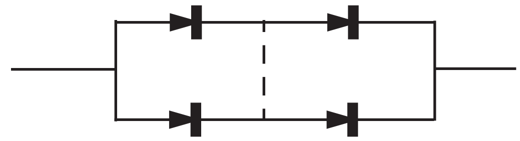
A serious problem with this superdiode is that it masks failures silently. There is no easy way to determine how much failure tolerance remains in the system.

8.4.3 Voting

Although there have been attempts to extend quad-component methods to digital logic, the intricacy of the required interconnections grows much too rapidly. Fortunately, there is a systematic alternative that takes advantage of the static discipline and level regeneration that are inherent properties of digital logic. In addition, it has the nice feature that it can be applied at any level of module, from a single gate on up to an entire computer. The technique is to substitute in place of a single module a set of replicas of that same module, all operating in parallel with the same inputs, and compare their outputs with a device known as a *voter*. This basic strategy is called *N-modular redundancy*, or NMR. When N has the value 3 the strategy is called *triple-modular redundancy*, abbreviated TMR. When other values are used for N the strategy is named by replacing the N of NMR with the number, as in 5MR. The combination of N replicas of some module and

FIGURE 8.7

A quad-component superdiode. The dotted line represents an optional bridging connection, which allows the superdiode to tolerate a different set of failures, as described in the text.



the voting system is sometimes called a *supermodule*. Several different schemes exist for interconnection and voting, only a few of which we explore here.

The simplest scheme, called *fail-vote,* consists of NMR with a majority voter. One assembles N replicas of the module and a voter that consists of an N-way comparator and some counting logic. If a majority of the replicas agree on the result, the voter accepts that result and passes it along to the next system component. If any replicas disagree with the majority, the voter may in addition raise an alert, calling for repair of the replicas that were in the minority. If there is no majority, the voter signals that the supermodule has failed. In failure-tolerance terms, a triply-redundant fail-vote supermodule can mask the failure of any one replica, and it is fail-fast if any two replicas fail in different ways.

If the reliability, as was defined in Section 8.2.2, of a single replica module is R and the underlying fault mechanisms are independent, a TMR fail-vote supermodule will operate correctly if all 3 modules are working (with reliability R^3) or if 1 module has failed and the other 2 are working (with reliability $R^2(1-R)$). Since a single-module failure can happen in 3 different ways, the reliability of the supermodule is the sum,

$$R_{supermodule} = R^3 + 3R^2(1-R) = 3R^2 - 2R^3 \qquad \text{Eq. 8–10}$$

but the supermodule is *not* always fail-fast. If two replicas fail in exactly the same way, the voter will accept the erroneous result and, unfortunately, call for repair of the one correctly operating replica. This outcome is not as unlikely as it sounds because several replicas that went through the same design and production process may have exactly the same set of design or manufacturing faults. This problem can arise despite the independence assumption used in calculating the probability of correct operation. That calculation assumes only that the probability that different replicas produce correct answers be independent; it assumes nothing about the probability of producing specific wrong answers. Without more information about the probability of specific errors and their correlations the only thing we can say about the probability that an incorrect result will be accepted by the voter is that it is not more than

$$(1-R_{supermodule}) = (1 - 3R^2 + 2R^3)$$

These calculations assume that the voter is perfectly reliable. Rather than trying to create perfect voters, the obvious thing to do is replicate them, too. In fact, everything—modules, inputs, outputs, sensors, actuators, etc.—should be replicated, and the final vote should be taken by the client of the system. Thus, three-engine airplanes vote with their propellers: when one engine fails, the two that continue to operate overpower the inoperative one. On the input side, the pilot's hand presses forward on three separate throttle levers. A fully replicated TMR supermodule is shown in Figure 8.8. With this interconnection arrangement, any measurement or estimate of the reliability, R, of a component module should include the corresponding voter. It is actually customary (and more logical) to consider a voter to be a component of the next module in the chain rather than, as the diagram suggests, the previous module. This fully replicated design is sometimes described as *recursive*.

The numerical effect of fail-vote TMR is impressive. If the reliability of a single module at time *T* is 0.999, equation 8–10 says that the reliability of a fail-vote TMR supermodule at that same time is 0.999997. TMR has reduced the probability of failure from one in a thousand to three in a million. This analysis explains why airplanes intended to fly across the ocean have more than one engine. Suppose that the rate of engine failures is such that a single-engine plane would fail to complete one out of a thousand trans-Atlantic flights. Suppose also that a 3-engine plane can continue flying as long as any 2 engines are operating, but it is too heavy to fly with only 1 engine. In 3 flights out of a thousand, one of the three engines will fail, but if engine failures are independent, in 999 out of each thousand first-engine failures, the remaining 2 engines allow the plane to limp home successfully.

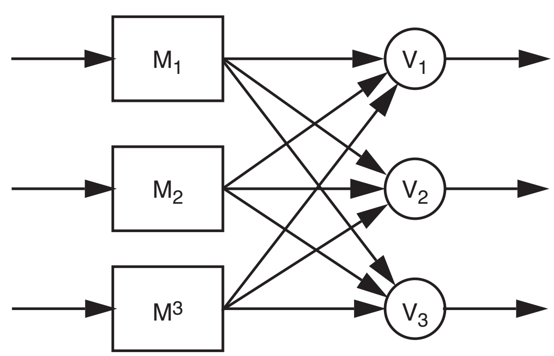
Although TMR has greatly improved reliability, it has not made a comparable impact on MTTF. In fact, the MTTF of a fail-vote TMR supermodule can be *smaller* than the MTTF of the original, single-replica module. The exact effect depends on the failure process of the replicas, so for illustration consider a memoryless failure process, not because it is realistic but because it is mathematically tractable. Suppose that airplane engines have an MTTF of 6,000 hours, they fail independently, the mechanism of engine failure is memoryless, and (since this is a fail-vote design) we need at least 2 operating engines to get home. When flying with three engines, the plane accumulates 6,000 hours of engine running time in only 2,000 hours of flying time, so from the point of view of the airplane as a whole, 2,000 hours is the expected time to the first engine failure. While flying with the remaining two engines, it will take another 3,000 flying hours to accumulate 6,000 more engine hours. Because the failure process is memoryless we can calculate the MTTF of the 3-engine plane by adding:

Mean time to first failure	2000 hours (three engines)
Mean time from first to second failure	3000 hours (two engines)
Total mean time to system failure	5000 hours

Thus the mean time to system failure is less than the 6,000 hour MTTF of a single engine. What is going on here is that we have actually sacrificed long-term reliability in order to enhance short-term reliability. Figure 8.9 illustrates the reliability of our hypo-

FIGURE 8.8

Triple-modular redundant supermodule, with three inputs, three voters, and three outputs.



thetical airplane during its 6 hours of flight, which amounts to only 0.001 of the single-engine MTTF—the mission time is very short compared with the MTTF and the reliability is far higher. Figure 8.10 shows the same curve, but for flight times that are comparable with the MTTF. In this region, if the plane tried to keep flying for 8000 hours (about 1.4 times the single-engine MTTF), a single-engine plane would fail to complete the flight in 3 out of 4 tries, but the 3-engine plane would fail to complete the flight in 5 out of 6 tries. (One should be wary of these calculations because the assumptions of independence and memoryless operation may not be met in practice. Sidebar 8.2 elaborates.)

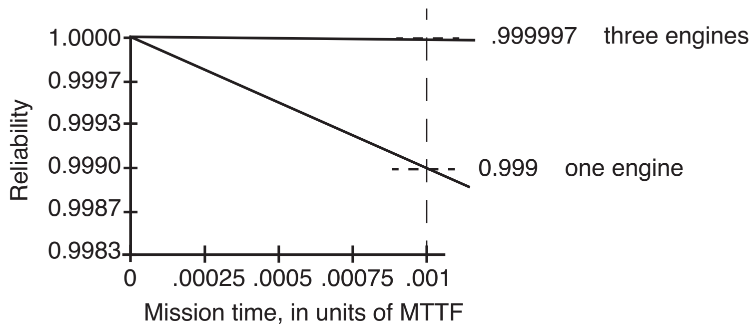


FIGURE 8.9

Reliability with triple modular redundancy, for mission times much less than the MTTF of 6,000 hours. The vertical dotted line represents a six-hour flight.

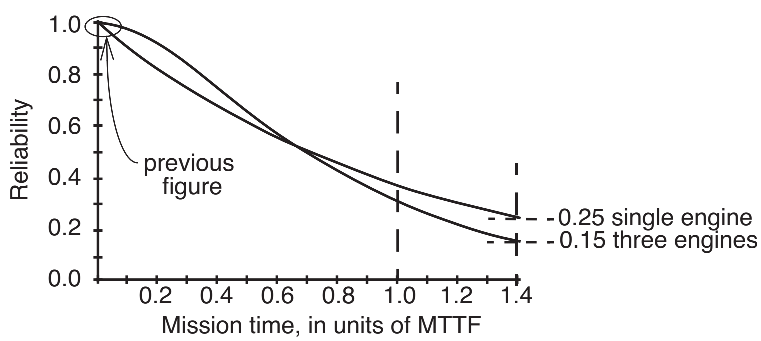


FIGURE 8.10

Reliability with triple modular redundancy, for mission times comparable to the MTTF of 6,000 hours. The two vertical dotted lines represent mission times of 6,000 hours (left) and 8,400 hours (right).

Sidebar 8.2: Risks of manipulating MTTFs The apparently casual manipulation of MTTFs in Sections 8.4.3 and 8.4.4 is justified by assumptions of independence of failures and memoryless processes. But one can trip up by blindly applying this approach without understanding its limitations. To see how, consider a computer system that has been observed for several years to have a hardware crash an average of every 2 weeks and a software crash an average of every 6 weeks. The operator does not repair the system, but simply restarts it and hopes for the best. The composite MTTF is 1.5 weeks, determined most easily by considering what happens if we run the system for, say, 60 weeks. During that time we expect to see

10 software failures
30 hardware failures
___
40 system failures in 60 weeks —> 1.5 weeks between failure

New hardware is installed, identical to the old except that it never fails. The MTTF should jump to 6 weeks because the only remaining failures are software, right?

Perhaps—but *only* if the software failure process is independent of the hardware failure process.

Suppose the software failure occurs because there is a bug (fault) in a clock-updating procedure: The bug always crashes the system exactly 420 hours (2 1/2 weeks) after it is started—if it gets a chance to run that long. The old hardware was causing crashes so often that the software bug only occasionally had a chance to do its thing—only about once every 6 weeks. Most of the time, the recovery from a hardware failure, which requires restarting the system, had the side effect of resetting the process that triggered the software bug. So, when the new hardware is installed, the system has an MTTF of only 2.5 weeks, much less than hoped.

MTTF's are useful, but one must be careful to understand what assumptions go into their measurement and use.

If we had assumed that the plane could limp home with just one engine, the MTTF would have increased, rather than decreased, but only modestly. Replication provides a dramatic improvement in reliability for missions of duration short compared with the MTTF, but the MTTF itself changes much less. We can verify this claim with a little more analysis, again assuming memoryless failure processes to make the mathematics tractable. Suppose we have an NMR system with the property that it somehow continues to be useful as long as at least one replica is still working. (This system requires using fail-fast replicas and a cleverer voter, as described in Section 8.4.4 below.) If a single replica has an $MTTF_{replica} = 1$, there are N independent replicas, and the failure process is memoryless, the expected time until the first failure is $MTTF_{replica}/N$, the expected time from then until the second failure is $MTTF_{replica}/(N-1)$, etc., and the expected time until the system of N replicas fails is the sum of these times,

$$MTTF_{system} = 1 + 1/2 + 1/3 + \ldots(1/N) \qquad \text{Eq. 8–11}$$

which for large N is approximately $ln(N)$. As we add to the cost by adding more replicas, $MTTF_{system}$ grows disappointingly slowly—proportional to the logarithm of the cost. To multiply the $MTTF_{system}$ by K, the number of replicas required is e^K —the cost grows exponentially. The significant conclusion is that *in systems for which the mission time is long compared with* $MTTF_{replica}$, *simple replication escalates the cost while providing little benefit*. On the other hand, there is a way of making replication effective for long missions, too. The method is to enhance replication by adding *repair*.

8.4.4 Repair

Let us return now to a fail-vote TMR supermodule (that is, it requires that at least two replicas be working) in which the voter has just noticed that one of the three replicas is producing results that disagree with the other two. Since the voter is in a position to report which replica has failed, suppose that it passes such a report along to a repair person who immediately examines the failing replica and either fixes or replaces it. For this approach, the mean time to repair (MTTR) measure becomes of interest. The supermodule fails if either the second or third replica fails before the repair to the first one can be completed. Our intuition is that if the MTTR is small compared with the combined MTTF of the other two replicas, the chance that the supermodule fails will be similarly small.

The exact effect on chances of supermodule failure depends on the shape of the reliability function of the replicas. In the case where the failure and repair processes are both memoryless, the effect is easy to calculate. Since the rate of failure of 1 replica is $1/MTTF$, the rate of failure of 2 replicas is $2/MTTF$. If the repair time is short compared with $MTTF$ the probability of a failure of 1 of the 2 remaining replicas while waiting a time T for repair of the one that failed is approximately $2T/MTTF$. Since the mean time to repair is MTTR, we have

$$Pr(\text{supermodule fails while waiting for repair}) = \frac{2 \times MTTR}{MTTF} \qquad \text{Eq. 8–12}$$

Continuing our airplane example and temporarily suspending disbelief, suppose that during a long flight we send a mechanic out on the airplane's wing to replace a failed engine. If the replacement takes 1 hour, the chance that one of the other two engines fails during that hour is approximately 1/3000. Moreover, once the replacement is complete, we expect to fly another 2000 hours until the next engine failure. Assuming further that the mechanic is carrying an unlimited supply of replacement engines, completing a 10,000 hour flight—or even a longer one—becomes plausible. The general formula for the MTTF of a fail-vote TMR supermodule with memoryless failure and repair processes is (this formula comes out of the analysis of continuous-transition birth-and-death Markov processes, an advanced probability technique that is beyond our scope):

$$MTTF_{\text{supermodule}} = \frac{MTTF_{replica}}{3} \times \frac{MTTF_{replica}}{2 \times MTTR_{replica}} = \frac{(MTTF_{replica})^2}{6 \times MTTR_{replica}} \qquad \text{Eq. 8–13}$$

Thus, our 3-engine plane with hypothetical in-flight repair has an MTTF of 6 million hours, an enormous improvement over the 6000 hours of a single-engine plane. This equation can be interpreted as saying that, compared with an unreplicated module, the MTTF has been reduced by the usual factor of 3 because there are 3 replicas, but at the same time the availability of repair has increased the MTTF by a factor equal to the ratio of the MTTF of the remaining 2 engines to the MTTR.

Replacing an airplane engine in flight may be a fanciful idea, but replacing a magnetic disk in a computer system on the ground is quite reasonable. Suppose that we store 3 replicas of a set of data on 3 independent hard disks, each of which has an MTTF of 5 years (using as the MTTF the expected operational lifetime, not the "MTTF" derived from the short-term failure rate). Suppose also, that if a disk fails, we can locate, install, and copy the data to a replacement disk in an average of 10 hours. In that case, by eq. 8–13, the MTTF of the data is

$$\frac{(MTTF_{replica})^2}{6 \times MTTR_{replica}} = \frac{(5\text{ years})^2}{6 \cdot (10\text{ hours})/(8760\text{ hours/year})} = 3650\text{ years} \qquad \text{Eq. 8–14}$$

In effect, redundancy plus repair has reduced the probability of failure of this supermodule to such a small value that for all practical purposes, failure can be neglected and the supermodule can operate indefinitely.

Before running out to start a company that sells superbly reliable disk-storage systems, it would be wise to review some of the overly optimistic assumptions we made in getting that estimate of the MTTF, most of which are not likely to be true in the real world:

- *Disks fail independently.* A batch of real world disks may all come from the same vendor, where they acquired the same set of design and manufacturing faults. Or, they may all be in the same machine room, where a single earthquake—which probably has an MTTF of less than 3,650 years—may damage all three.
- *Disk failures are memoryless.* Real-world disks follow a bathtub curve. If, when disk #1 fails, disk #2 has already been in service for three years, disk #2 no longer has an expected operational lifetime of 5 years, so the chance of a second failure while waiting for repair is higher than the formula assumes. Furthermore, when disk #1 is replaced, its chances of failing are probably higher than usual for the first few weeks.
- *Repair is also a memoryless process.* In the real world, if we stock enough spares that we run out only once every 10 years and have to wait for a shipment from the factory, but doing a replacement happens to run us out of stock today, we will probably still be out of stock tomorrow and the next day.
- *Repair is done flawlessly.* A repair person may replace the wrong disk, forget to copy the data to the new disk, or install a disk that hasn't passed burn-in and fails in the first hour.

Each of these concerns acts to reduce the reliability below what might be expected from our overly simple analysis. Nevertheless, NMR with repair remains a useful technique, and in Chapter 10[on-line] we will see ways in which it can be applied to disk storage.

One of the most powerful applications of NMR is in the masking of transient errors. When a transient error occurs in one replica, the NMR voter immediately masks it. Because the error is transient, the subsequent behavior of the supermodule is as if repair happened by the next operation cycle. The numerical result is little short of extraordinary. For example, consider a processor arithmetic logic unit (ALU) with a 1 gigahertz clock and which is triply replicated with voters checking its output at the end of each clock cycle. In equation 8–13 we have $MTTR_{replica} = 1$ (in this application, equation 8–13 is only an approximation because the time to repair is a constant rather than the result of a memoryless process), and $MTTF_{supermodule} = (MTTF_{replica})^2/6$ cycles. If $MTTF_{replica}$ is 10^{10} cycles (1 error in 10 billion cycles, which at this clock speed means one error every 10 seconds), $MTTF_{supermodule}$ is $10^{20}/6$ cycles, about 500 years. TMR has taken three ALUs that were for practical use nearly worthless and created a super-ALU that is almost infallible.

The reason things seem so good is that we are evaluating the chance that two transient errors occur in the same operation cycle. If transient errors really are independent, that chance is small. This effect is powerful, but the leverage works in both directions, thereby creating a potential hazard: it is especially important to keep track of the rate at which transient errors actually occur. If they are happening, say, 20 times as often as hoped, $MTTF_{supermodule}$ will be 1/400 of the original prediction—the super-ALU is likely to fail once per year. That may still be acceptable for some applications, but it is a big change. Also, as usual, the assumption of independence is absolutely critical. If all the ALUs came from the same production line, it seems likely that they will have at least some faults in common, in which case the super-ALU may be just as worthless as the individual ALUs.

Several variations on the simple fail-vote structure appear in practice:

- *Purging*. In an NMR design with a voter, whenever the voter detects that one replica disagrees with the majority, the voter calls for its repair and in addition marks that replica DOWN and ignores its output until hearing that it has been repaired. This technique doesn't add anything to a TMR design, but with higher levels of replication, as long as replicas fail one at a time and any two replicas continue to operate correctly, the supermodule works.

- *Pair-and-compare*. Create a fail-fast module by taking two replicas, giving them the same inputs, and connecting a simple comparator to their outputs. As long as the comparator reports that the two replicas of a pair agree, the next stage of the system accepts the output. If the comparator detects a disagreement, it reports that the module has failed. The major attraction of pair-and-compare is that it can be used to create fail-fast modules starting with easily available commercial, off-the-shelf components, rather than commissioning specialized fail-fast versions. Special high-reliability components typically have a cost that is much higher than off-the-shelf designs, for two reasons. First, since they take more time to design and test, the ones that are available are typically of an older, more expensive technology. Second, they are usually low-volume products that cannot take advantage of economies of large-scale production. These considerations also conspire to produce long delivery cycles, making it harder to keep spares in stock. An important aspect of using standard, high-volume, low-cost components is that one can afford to keep a stock of spares, which in turn means that MTTR can be made small: just replace a failing replica with a spare (the popular term for this approach is *pair-and-spare*) and do the actual diagnosis and repair at leisure.

- *NMR with fail-fast replicas.* If each of the replicas is itself a fail-fast design (perhaps using pair-and-compare internally), then a voter can restrict its attention to the outputs of only those replicas that claim to be producing good results and ignore those that are reporting that their outputs are questionable. With this organization, a TMR system can continue to operate even if 2 of its 3 replicas have failed, since the 1 remaining replica is presumably checking its own results. An NMR system with repair and constructed of fail-fast replicas is so robust that it is unusual to find examples for which N is greater than 2.

Figure 8.11 compares the ability to continue operating until repair arrives of 5MR designs that use fail-vote, purging, and fail-fast replicas. The observant reader will note that this chart can be deemed guilty of a misleading comparison, since it claims that the 5MR system continues working when only one fail-fast replica is still running. But if that fail-fast replica is actually a pair-and-compare module, it might be more accurate to say that there are two still-working replicas at that point.

Another technique that takes advantage of repair, can improve availability, and can degrade gracefully (in other words, it can be fail-soft) is called *partition*. If there is a choice of purchasing a system that has either one fast processor or two slower processors, the two-processor system has the virtue that when one of its processors fails, the system

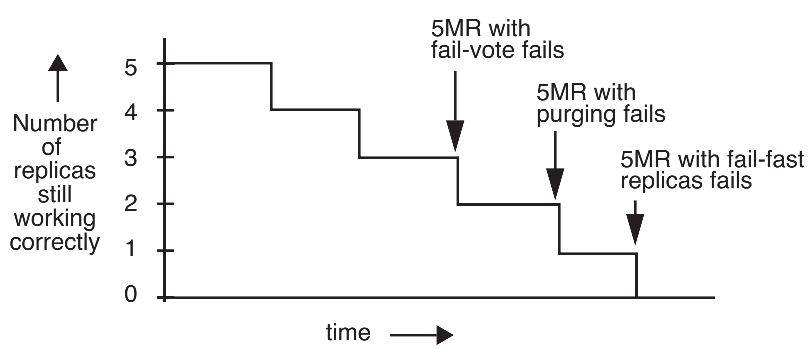


FIGURE 8.11

Failure points of three different 5MR supermodule designs, if repair does not happen in time.

can continue to operate with half of its usual capacity until someone can repair the failed processor. An electric power company, rather than installing a single generator of capacity K megawatts, may install N generators of capacity K/N megawatts each.

When equivalent modules can easily share a load, partition can extend to what is called $N + 1$ *redundancy*. Suppose a system has a load that would require the capacity of N equivalent modules. The designer partitions the load across $N + 1$ or more modules. Then, if any one of the modules fails, the system can carry on at full capacity until the failed module can be repaired.

$N + 1$ redundancy is most applicable to modules that are completely interchangeable, can be dynamically allocated, and are not used as storage devices. Examples are processors, dial-up modems, airplanes, and electric generators. Thus, one extra airplane located at a busy hub can mask the failure of any single plane in an airline's fleet. When modules are not completely equivalent (for example, electric generators come in a range of capacities, but can still be interconnected to share load), the design must ensure that the spare capacity is greater than the capacity of the largest individual module. For devices that provide storage, such as a hard disk, it is also possible to apply partition and $N + 1$ redundancy with the same goals, but it requires a greater level of organization to preserve the stored contents when a failure occurs, for example by using RAID, as was described in Section 8.4.1, or some more general replica management system such as those discussed in Section 10.3.7.

For some applications an occasional interruption of availability is acceptable, while in others every interruption causes a major problem. When repair is part of the fault tolerance plan, it is sometimes possible, with extra care and added complexity, to design a system to provide *continuous operation.* Adding this feature requires that when failures occur, one can quickly identify the failing component, remove it from the system, repair it, and reinstall it (or a replacement part) all without halting operation of the system. The design required for continuous operation of computer hardware involves connecting and disconnecting cables and turning off power to some components but not others, without damaging anything. When hardware is designed to allow connection and disconnection from a system that continues to operate, it is said to allow *hot swap*.

In a computer system, continuous operation also has significant implications for the software. Configuration management software must anticipate hot swap so that it can stop using hardware components that are about to be disconnected, as well as discover newly attached components and put them to work. In addition, maintaining state is a challenge. If there are periodic consistency checks on data, those checks (and repairs to data when the checks reveal inconsistencies) must be designed to work correctly even though the system is in operation and the data is perhaps being read and updated by other users at the same time.

Overall, continuous operation is not a feature that should be casually added to a list of system requirements. When someone suggests it, it may be helpful to point out that it is much like trying to keep an airplane flying indefinitely. Many large systems that appear to provide continuous operation are actually designed to stop occasionally for maintenance.

8.5 Applying Redundancy to Software and Data

The examples of redundancy and replication in the previous sections all involve hardware. A seemingly obvious next step is to apply the same techniques to software and to data. In the case of software the goal is to reduce the impact of programming errors, while in the case of data the goal is to reduce the impact of any kind of hardware, software, or operational error that might affect its integrity. This section begins the exploration of several applicable techniques: N-version programming, valid construction, and building a firewall to separate stored state into two categories: state whose integrity must be preserved and state that can casually be abandoned because it is easy to reconstruct.

8.5.1 Tolerating Software Faults

Simply running three copies of the same buggy program is likely to produce three identical incorrect results. NMR requires independence among the replicas, so the designer needs a way of introducing that independence. An example of a way of introducing independence is found in the replication strategy for the root name servers of the Internet Domain Name System (DNS, described in Section 4.4). Over the years, slightly different implementations of the DNS software have evolved for different operating systems, so the root name server replicas intentionally employ these different implementations to reduce the risk of replicated errors.

To try to harness this idea more systematically, one can commission several teams of programmers and ask each team to write a complete version of an application according to a single set of specifications. Then, run these several versions in parallel and compare their outputs. The hope is that the inevitable programming errors in the different versions will be independent and voting will produce a reliable system. Experiments with this technique, known as *N-version programming*, suggest that the necessary independence is hard to achieve. Different programmers may be trained in similar enough ways that they make the same mistakes. Use of the same implementation language may encourage the same errors. Ambiguities in the specification may be misinterpreted in the same way by more than one team and the specification itself may contain errors. Finally, it is hard to write a specification in enough detail that the outputs of different implementations can be expected to be bit-for-bit identical. The result is that after much effort, the technique may still mask only a certain class of bugs and leave others unmasked. Nevertheless, there are reports that *N*-version programming has been used, apparently with success, in at least two safety-critical aerospace systems, the flight control system of the Boeing 777 aircraft (with $N = 3$) and the on-board control system for the Space Shuttle (with $N = 2$).

Incidentally, the strategy of employing multiple design teams can also be applied to hardware replicas, with a goal of increasing the independence of the replicas by reducing the chance of replicated design errors and systematic manufacturing defects.

Much of software engineering is devoted to a different approach: devising specification and programming techniques that avoid faults in the first place and test techniques that systematically root out faults so that they can be repaired once and for all before deploying the software. This approach, sometimes called *valid construction*, can dramatically reduce the number of software faults in a delivered system, but because it is difficult both to completely specify and to completely test a system, some faults inevitably remain. Valid construction is based on the observation that software, unlike hardware, is not subject to wear and tear, so if it is once made correct, it should stay that way. Unfortunately, this observation can turn out to be wishful thinking, first because it is hard to make software correct, and second because it is nearly always necessary to make changes after installing a program because the requirements, the environment surrounding the program, or both, have changed. There is thus a potential for tension between valid construction and the principle that one should *design for iteration*.

Worse, later maintainers and reworkers often do not have a complete understanding of the ground rules that went into the original design, so their work is likely to introduce new faults for which the original designers did not anticipate providing tests. Even if the original design is completely understood, when a system is modified to add features that were not originally planned, the original ground rules may be subjected to some violence. Software faults more easily creep into areas that lack systematic design.

8.5.2 Tolerating Software (and other) Faults by Separating State

Designers of reliable systems usually assume that, despite the best efforts of programmers there will always be a residue of software faults, just as there is also always a residue of hardware, operation, and environment faults. The response is to develop a strategy for tolerating all of them. Software adds the complication that the current state of a running program tends to be widely distributed. Parts of that state may be in non-volatile storage, while other parts are in temporary variables held in volatile memory locations, processor registers, and kernel tables. This wide distribution of state makes containment of errors problematic. As a result, when an error occurs, any strategy that involves stopping some collection of running threads, tinkering to repair the current state (perhaps at the same time replacing a buggy program module), and then resuming the stopped threads is usually unrealistic.

In the face of these observations, a programming discipline has proven to be effective: systematically divide the current state of a running program into two mutually exclusive categories and separate the two categories with a firewall. The two categories are:

- State that the system can safely abandon in the event of a failure.
- State whose integrity the system should preserve despite failure.

Upon detecting a failure, the plan becomes to abandon all state in the first category and instead concentrate just on maintaining the integrity of the data in the second category. An important part of the strategy is an important *sweeping simplification*: classify the state of running threads (that is, the thread table, stacks, and registers) as abandonable. When a failure occurs, the system abandons the thread or threads that were running at the time and instead expects a restart procedure, the system operator, or the individual user to start a new set of threads with a clean slate. The new thread or threads can then, working with only the data found in the second category, verify the integrity of that data and return to normal operation. The primary challenge then becomes to build a firewall that can protect the integrity of the second category of data despite the failure.

The designer can base a natural firewall on the common implementations of volatile (e.g., CMOS memory) and non-volatile (e.g., magnetic disk) storage. As it happens, writing to non-volatile storage usually involves mechanical movement such as rotation of a disk platter, so most transfers move large blocks of data to a limited region of addresses, using a GET/PUT interface. On the other hand, volatile storage technologies typically provide a READ/WRITE interface that allows rapid-fire writes to memory addresses chosen at random, so failures that originate in or propagate to software tend to quickly and untraceably corrupt random-access data. By the time an error is detected the software may thus have already damaged a large and unidentifiable part of the data in volatile memory. The GET/PUT interface instead acts as a bottleneck on the rate of spread of data corruption. The goal can be succinctly stated: to detect failures and stop the system before it reaches the next PUT operation, thus making the volatile storage medium the error containment boundary. It is only incidental that volatile storage usually has a READ/WRITE interface, while non-volatile storage usually has a GET/PUT interface, but because that is usually true it becomes a convenient way to implement and describe the firewall.

This technique is widely used in systems whose primary purpose is to manage long-lived data. In those systems, two aspects are involved:

- Prepare for failure by recognizing that all state in volatile memory devices can vanish at any instant, without warning. When it does vanish, automatically launch new threads that start by restoring the data in non-volatile storage to a consistent, easily described state. The techniques to do this restoration are called *recovery.* Doing recovery systematically involves atomicity, which is explored in Chapter 9[on-line].
- Protect the data in non-volatile storage using replication, thus creating the class of storage known as *durable* storage. Replicating data can be a straightforward application of redundancy, so we will begin the topic in this chapter. However, there are more effective designs that make use of atomicity and geographical separation of replicas, so we will revisit durability in Chapter 10[on-line].

When the volatile storage medium is CMOS RAM and the non-volatile storage medium is magnetic disk, following this programming discipline is relatively straightforward because the distinctively different interfaces make it easy to remember where to place data. But when a one-level store is in use, giving the appearance of random access to all storage, or the non-volatile medium is flash memory, which allows fast random access, it may be necessary for the designer to explicitly specify both the firewall mechanism and which data items are to reside on each side of the firewall.

A good example of the firewall strategy can be found in most implementations of Internet Domain Name System servers. In a typical implementation the server stores the authoritative name records for its domain on magnetic disk, and copies those records into volatile CMOS memory either at system startup or the first time it needs a particular record. If the server fails for any reason, it simply abandons the volatile memory and restarts. In some implementations, the firewall is reinforced by not having any PUT operations in the running name server. Instead, the service updates the authoritative name records using a separate program that runs when the name server is off-line.

In addition to employing independent software implementations and a firewall between categories of data, DNS also protects against environmental faults by employing geographical separation of its replicas, a topic that is explored more deeply in Section 10.3[on-line]. The three techniques taken together make DNS quite fault tolerant.

8.5.3 Durability and Durable Storage

For the discipline just described to work, we need to make the result of a PUT operation durable. But first we must understand just what "durable" means. *Durability* is a specification of how long the result of an action must be preserved after the action completes. One must be realistic in specifying durability because there is no such thing as perfectly durable storage in which the data will be remembered forever. However, by choosing enough genuinely independent replicas, and with enough care in management, one can meet any reasonable requirement.

Durability specifications can be roughly divided into four categories, according to the length of time that the application requires that data survive. Although there are no bright dividing lines, as one moves from one category to the next the techniques used to achieve durability tend to change.

- *Durability no longer than the lifetime of the thread that created the data.* For this case, it is usually adequate to place the data in volatile memory.

For example, an action such as moving the gearshift may require changing the operating parameters of an automobile engine. The result must be reliably remembered, but only until the next shift of gears or the driver switches the engine off.

The operations performed by calls to the kernel of an operating system provide another example. The CHDIR procedure of the UNIX kernel (see Table 2.1 in Section 2.5.1) changes the working directory of the currently running process. The kernel state variable that holds the name of the current working directory is a value in volatile RAM that does not need to survive longer than this process.

For a third example, the registers and cache of a hardware processor usually provide just the first category of durability. If there is a failure, the plan is to abandon those values along with the contents of volatile memory, so there is no need for a higher level of durability.

- *Durability for times short compared with the expected operational lifetime of non-volatile storage media such as magnetic disk or flash memory.* A designer typically implements this category of durability by writing one copy of the data in the non-volatile storage medium.

Returning to the automotive example, there may be operating parameters such as engine timing that, once calibrated, should be durable at least until the next tune-up, not just for the life of one engine use session. Data stored in a cache that writes through to a non-volatile medium has about this level of durability. As a third example, a remote procedure call protocol that identifies duplicate messages by recording nonces might write old nonce values (see Section 7.5.3) to a non-volatile storage medium, knowing that the real goal is not to remember the nonces forever, but rather to make sure that the nonce record outlasts the longest retry timer of any client. Finally, text editors and word-processing systems typically write temporary copies on magnetic disk of the material currently being edited so that if there is a system crash or power failure the user does not have to repeat the entire editing session. These temporary copies need to survive only until the end of the current editing session.

- *Durability for times comparable to the expected operational lifetime of non-volatile storage media.* Because actual non-volatile media lifetimes vary quite a bit around the expected lifetime, implementation generally involves placing replicas of the data on independent instances of the non-volatile media.

This category of durability is the one that is usually called *durable storage* and it is the category for which the next section of this chapter develops techniques for implementation. Users typically expect files stored in their file systems and data managed by a database management system to have this level of durability. Section 10.3[on-line] revisits the problem of creating durable storage when replicas are geographically separated.

- *Durability for many multiples of the expected operational lifetime of non-volatile storage media.*

This highest level of durability is known as *preservation*, and is the specialty of archivists. In addition to making replicas and keeping careful records, it involves copying data from one non-volatile medium to another before the first one deteriorates or becomes obsolete. Preservation also involves (sometimes heroic) measures to preserve the ability to correctly interpret idiosyncratic formats created by software that has long since become obsolete. Although important, it is a separate topic, so preservation is not discussed any further here.

8.5.4 Magnetic Disk Fault Tolerance

In principle, durable storage can be constructed starting with almost any storage medium, but it is most straightforward to use non-volatile devices. Magnetic disks (see Sidebar 2.8) are widely used as the basis for durable storage because of their low cost, large capacity and non-volatility—they retain their memory when power is turned off or is accidentally disconnected. Even if power is lost during a write operation, at most a small block of data surrounding the physical location that was being written is lost, and disks can be designed with enough internal power storage and data buffering to avoid even that loss. In its raw form, a magnetic disk is remarkably reliable, but it can still fail in various ways and much of the complexity in the design of disk systems consists of masking these failures.

Conventionally, magnetic disk systems are designed in three nested layers. The innermost layer is the spinning disk itself, which provides what we will call *raw storage*. The next layer is a combination of hardware and firmware of the disk controller that provides for detecting the failures in the raw storage layer; it creates *fail-fast storage*. Finally, the hard disk firmware adds a third layer that takes advantage of the detection features of the second layer to create a substantially more reliable storage system, known as *careful storage*. Most disk systems stop there, but high-availability systems add a fourth layer to create *durable storage*. This section develops a disk failure model and explores error masking techniques for all four layers.

In early disk designs, the disk controller presented more or less the raw disk interface, and the fail-fast and careful layers were implemented in a software component of the operating system called the disk driver. Over the decades, first the fail-fast layer and more recently part or all of the careful layer of disk storage have migrated into the firmware of the disk controller to create what is known in the trade as a "hard drive". A hard drive usually includes a RAM buffer to hold a copy of the data going to and from the disk, both to avoid the need to match the data rate to and from the disk head with the data rate to and from the system memory and also to simplify retries when errors occur. RAID systems, which provide a form of durable storage, generally are implemented as an additional hardware layer that incorporates mass-market hard drives. One reason for this move of error masking from the operating system into the disk controller is that as computational power has gotten cheaper, the incremental cost of a more elaborate firmware design has dropped. A second reason may explain the obvious contrast with the lack of enthusiasm for memory parity checking hardware that is mentioned in Section 8.8.1. A transient memory error is all but indistinguishable from a program error, so the hardware vendor is not likely to be blamed for it. On the other hand, most disk errors have an obvious source, and hard errors are not transient. Because blame is easy to place, disk vendors have a strong motivation to include error masking in their designs.

8.5.4.1 Magnetic Disk Fault Modes

Sidebar 2.8 described the physical design of the magnetic disk, including platters, magnetic material, read/write heads, seek arms, tracks, cylinders, and sectors, but it did not make any mention of disk reliability. There are several considerations:

- Disks are high precision devices made to close tolerances. Defects in manufacturing a recording surface typically show up in the field as a sector that does not reliably record data. Such defects are a source of hard errors. Deterioration of the surface of a platter with age can cause a previously good sector to fail. Such loss is known as *decay* and, since any data previously recorded there is lost forever, decay is another example of hard error.

- Since a disk is mechanical, it is subject to wear and tear. Although a modern disk is a sealed unit, deterioration of its component materials as they age can create dust. The dust particles can settle on a magnetic surface, where they may interfere either with reading or writing. If interference is detected, then re-reading or re-writing that area of the surface, perhaps after jiggling the seek arm back and forth, may succeed in getting past the interference, so the fault may be transient. Another source of transient faults is electrical noise spikes. Because disk errors caused by transient faults can be masked by retry, they fall in the category of soft errors.
- If a running disk is bumped, the shock may cause a head to hit the surface of a spinning platter, causing what is known as a head crash. A head crash not only may damage the head and destroy the data at the location of impact, it also creates a cloud of dust that interferes with the operation of heads on other platters. A head crash generally results in several sectors decaying simultaneously. A set of sectors that tend to all fail together is known as a *decay set*. A decay set may be quite large, for example all the sectors on one drive or on one disk platter.
- As electronic components in the disk controller age, clock timing and signal detection circuits can go out of tolerance, causing previously good data to become unreadable, or bad data to be written, either intermittently or permanently. In consequence, electronic component tolerance problems can appear either as soft or hard errors.
- The mechanical positioning systems that move the seek arm and that keep track of the rotational position of the disk platter can fail in such a way that the heads read or write the wrong track or sector within a track. This kind of fault is known as a *seek error*.

8.5.4.2 System Faults

In addition to failures within the disk subsystem, there are at least two threats to the integrity of the data on a disk that arise from outside the disk subsystem:

- If the power fails in the middle of a disk write, the sector being written may end up being only partly updated. After the power is restored and the system restarts, the next reader of that sector may find that the sector begins with the new data, but ends with the previous data.
- If the operating system fails during the time that the disk is writing, the data being written could be affected, even if the disk is perfect and the rest of the system is fail-fast. The reason is that all the contents of volatile memory, including the disk buffer, are inside the fail-fast error containment boundary and thus at risk of damage when the system fails. As a result, the disk channel may correctly write on the disk what it reads out of the disk buffer in memory, but the faltering operating system may have accidentally corrupted the contents of that buffer after the

application called PUT. In such cases, the data that ends up on the disk will be corrupted, but there is no sure way in which the disk subsystem can detect the problem.

8.5.4.3 Raw Disk Storage

Our goal is to devise systematic procedures to mask as many of these different faults as possible. We start with a model of disk operation from a programmer's point of view. The raw disk has, at least conceptually, a relatively simple interface: There is an operation to seek to a (numbered) track, an operation that writes data on the track and an operation that reads data from the track. The failure model is simple: all errors arising from the failures just described are untolerated. (In the procedure descriptions, arguments are call-by-reference, and GET operations read from the disk into the argument named *data*.)
The raw disk layer implements these storage access procedures and failure tolerance model:

```
RAW_SEEK (track)    // Move read/write head into position.
RAW_PUT (data)      // Write entire track.
RAW_GET (data)      // Read entire track.
```

- error-free operation: RAW_SEEK moves the seek arm to position *track*. RAW_GET returns whatever was most recently written by RAW_PUT at position *track*.
- untolerated error: On any given attempt to read from or write to a disk, dust particles on the surface of the disk or a temporarily high noise level may cause data to be read or written incorrectly. (soft error)
- untolerated error: A spot on the disk may be defective, so all attempts to write to any track that crosses that spot will be written incorrectly. (hard error)
- untolerated error: Information previously written correctly may decay, so RAW_GET returns incorrect data. (hard error)
- untolerated error: When asked to read data from or write data to a specified track, a disk may correctly read or write the data, but on the wrong track. (seek error)
- untolerated error: The power fails during a RAW_PUT with the result that only the first part of *data ends* up being written on *track*. The remainder of *track* may contain older data.
- untolerated error: The operating system crashes during a RAW_PUT and scribbles over the disk buffer in volatile storage, so RAW_PUT writes corrupted data on one track of the disk.

8.5.4.4 Fail-Fast Disk Storage

The fail-fast layer is the place where the electronics and microcode of the disk controller divide the raw disk track into sectors. Each sector is relatively small, individually protected with an error-detection code, and includes in addition to a fixed-sized space for data a sector and track number. The error-detection code enables the disk controller to return a status code on FAIL_FAST_GET that tells whether a sector read correctly or incorrectly, and the sector and track numbers enable the disk controller to verify that the seek ended up on the correct track. The FAIL_FAST_PUT procedure not only writes the data, but it verifies that the write was successful by reading the newly written sector on the next rotation and comparing it with the data still in the write buffer. The sector thus becomes the minimum unit of reading and writing, and the disk address becomes the pair {*track*, *sector_number*}. For performance enhancement, some systems allow the caller to bypass the verification step of FAIL_FAST_PUT. When the client chooses this bypass, write failures become indistinguishable from decay events.

There is always a possibility that the data on a sector is corrupted in such a way that the error-detection code accidentally verifies. For completeness, we will identify that case as an untolerated error, but point out that the error-detection code should be powerful enough that the probability of this outcome is negligible.

The fail-fast layer implements these storage access procedures and failure tolerance model:

status ← FAIL_FAST_SEEK (*track*)
status ← FAIL_FAST_PUT (*data*, *sector_number*)
status ← FAIL_FAST_GET (*data*, *sector_number*)

- error-free operation: FAIL_FAST_SEEK moves the seek arm to *track*. FAIL_FAST_GET returns whatever was most recently written by FAIL_FAST_PUT at *sector_number* on *track* and returns *status* = OK.
- detected error: FAIL_FAST_GET reads the data, checks the error-detection code and finds that it does not verify. The cause may a soft error, a hard error due to decay, or a hard error because there is a bad spot on the disk and the invoker of a previous FAIL_FAST_PUT chose to bypass verification. FAIL_FAST_GET does not attempt to distinguish these cases; it simply reports the error by returning *status* = BAD.
- detected error: FAIL_FAST_PUT writes the data, on the next rotation reads it back, checks the error-detection code, finds that it does not verify, and reports the error by returning *status* = BAD.
- detected error: FAIL_FAST_SEEK moves the seek arm, reads the permanent track number in the first sector that comes by, discovers that it does not match the requested track number (or that the sector checksum does not verify), and reports the error by returning *status* = BAD.
- detected error: The caller of FAIL_FAST_PUT tells it to bypass the verification step, so FAIL_FAST_PUT always reports *status* = OK even if the sector was not written correctly. But a later caller of FAIL_FAST_GET that requests that sector should detect any such error.
- detected error: The power fails during a FAIL_FAST_PUT with the result that only the first part of *data ends* up being written on *sector*. The remainder of *sector* may contain older data. Any later call of FAIL_FAST_GET for that sector should discover that the sector checksum fails to verify and will thus return *status* = BAD.

Many (but not all) disks are designed to mask this class of failure by maintaining a reserve of power that is sufficient to complete any current sector write, in which case loss of power would be a tolerated failure.

- untolerated error: The operating system crashes during a FAIL_FAST_PUT and scribbles over the disk buffer in volatile storage, so FAIL_FAST_PUT writes corrupted data on one sector of the disk.
- untolerated error: The data of some sector decays in a way that is undetectable—the checksum accidentally verifies. (Probability should be negligible.)

8.5.4.5 Careful Disk Storage

The fail-fast disk layer detects but does not mask errors. It leaves masking to the careful disk layer, which is also usually implemented in the firmware of the disk controller. The careful layer checks the value of *status* following each disk SEEK, GET and PUT operation, retrying the operation several times if necessary, a procedure that usually recovers from seek errors and soft errors caused by dust particles or a temporarily elevated noise level. Some disk controllers seek to a different track and back in an effort to dislodge the dust. The careful storage layer implements these storage procedures and failure tolerance model:

status ← CAREFUL_SEEK (*track*)
status ← CAREFUL_PUT (*data*, *sector_number*)
status ← CAREFUL_GET (*data*, *sector_number*)

- error-free operation: CAREFUL_SEEK moves the seek arm to *track*. CAREFUL_GET returns whatever was most recently written by CAREFUL_PUT at *sector_number* on *track*. All three return *status* = OK.
- tolerated error: *Soft read, write, or seek error.* CAREFUL_SEEK, CAREFUL_GET and CAREFUL_PUT mask these errors by repeatedly retrying the operation until the fail-fast layer stops detecting an error, returning with *status* = OK. The careful storage layer counts the retries, and if the retry count exceeds some limit, it gives up and declares the problem to be a hard error.
- detected error: *Hard error.* The careful storage layer distinguishes hard from soft errors by their persistence through several attempts to read, write, or seek, and reports them to the caller by setting *status* = BAD. (But also see the note on *revectoring* below.)
- detected error: The power fails during a CAREFUL_PUT with the result that only the first part of *data* ends up being written on *sector*. The remainder of *sector* may contain older data. Any later call of CAREFUL_GET for that sector should discover that the sector checksum fails to verify and will thus return *status* = BAD. (Assuming that the fail-fast layer does not tolerate power failures.)
- untolerated error: *Crash corrupts data.* The system crashes during CAREFUL_PUT and corrupts the disk buffer in volatile memory, so CAREFUL_PUT correctly writes to the

disk sector the corrupted data in that buffer. The sector checksum of the fail-fast layer cannot detect this case.

- untolerated error: The data of some sector decays in a way that is undetectable—the checksum accidentally verifies. (Probability should be negligible)

Figure 8.12 exhibits algorithms for CAREFUL_GET and CAREFUL_PUT. The procedure CAREFUL_GET, by repeatedly reading any data with *status* = BAD, masks soft read errors. Similarly, CAREFUL_PUT retries repeatedly if the verification done by FAIL_FAST_PUT fails, thereby masking soft write errors, whatever their source.

The careful layer of most disk controller designs includes one more feature: if CAREFUL_PUT detects a hard error while writing a sector, it may instead write the data on a spare sector elsewhere on the same disk and add an entry to an internal disk mapping table so that future GETs and PUTs that specify that sector instead use the spare. This mechanism is called *revectoring*, and most disk designs allocate a batch of spare sectors for this purpose. The spares are not usually counted in the advertised disk capacity, but the manufacturer's advertising department does not usually ignore the resulting increase in the expected operational lifetime of the disk. For clarity of the discussion we omit that feature.

As indicated in the failure tolerance analysis, there are still two modes of failure that remain unmasked: a crash during CAREFUL_PUT may undetectably corrupt one disk sector, and a hard error arising from a bad spot on the disk or a decay event may detectably corrupt any number of disk sectors.

8.5.4.6 Durable Storage: RAID 1

For durability, the additional requirement is to mask decay events, which the careful storage layer only detects. The primary technique is that the PUT procedure should write several replicas of the data, taking care to place the replicas on different physical devices with the hope that the probability of disk decay in one replica is independent of the prob-

```
procedure CAREFUL_GET (data, sector_number)
    for i from 1 to NTRIES do
        if FAIL_FAST_GET (data, sector_number) = OK then
            return OK
    return BAD

procedure CAREFUL_PUT (data, sector_number)
    for i from 1 to NTRIES do
        if FAIL_FAST_PUT (data, sector_number) = OK then
            return OK
    return BAD
```

FIGURE 8.12

Procedures that implement careful disk storage.

ability of disk decay in the next one, and the number of replicas is large enough that when a disk fails there is enough time to replace it before all the other replicas fail. Disk system designers call these replicas *mirrors*. A carefully designed replica strategy can create storage that guards against premature disk failure and that is durable enough to substantially exceed the expected operational lifetime of any single physical disk. Errors on reading are detected by the fail-fast layer, so it is not usually necessary to read more than one copy unless that copy turns out to be bad. Since disk operations may involve more than one replica, the track and sector numbers are sometimes encoded into a virtual sector number and the durable storage layer automatically performs any needed seeks.

The durable storage layer implements these storage access procedures and failure tolerance model:

status ← DURABLE_PUT (*data*, *virtual_sector_number*)
status ← DURABLE_GET (*data*, *virtual_sector_number*)

- error-free operation: DURABLE_GET returns whatever was most recently written by DURABLE_PUT at *virtual_sector_number* with *status* = OK.
- tolerated error: Hard errors reported by the careful storage layer are masked by reading from one of the other replicas. The result is that the operation completes with *status* = OK.
- untolerated error: A decay event occurs on the same sector of all the replicas, and the operation completes with *status* = BAD.
- untolerated error: The operating system crashes during a DURABLE_PUT and scribbles over the disk buffer in volatile storage, so DURABLE_PUT writes corrupted data on all mirror copies of that sector.
- untolerated error: The data of some sector decays in a way that is undetectable—the checksum accidentally verifies. (Probability should be negligible)

In this accounting there is no mention of soft errors or of positioning errors because they were all masked by a lower layer.

One configuration of RAID (see Section 2.1.1.4), known as "RAID 1", implements exactly this form of durable storage. RAID 1 consists of a tightly-managed array of identical replica disks in which DURABLE_PUT (*data*, *sector_number*) writes *data* at the same *sector_number* of each disk and DURABLE_GET reads from whichever replica copy has the smallest expected latency, which includes queuing time, seek time, and rotation time. With RAID, the decay set is usually taken to be an entire hard disk. If one of the disks fails, the next DURABLE_GET that tries to read from that disk will detect the failure, mask it by reading from another replica, and put out a call for repair. Repair consists of first replacing the disk that failed and then copying all of the disk sectors from one of the other replica disks.

8.5.4.7 Improving on RAID 1

Even with RAID 1, an untolerated error can occur if a rarely-used sector decays, and before that decay is noticed all other copies of that same sector also decay. When there is finally a call for that sector, all fail to read and the data is lost. A closely related scenario is that a sector decays and is eventually noticed, but the other copies of that same sector decay before repair of the first one is completed. One way to reduce the chances of these outcomes is to implement a clerk that periodically reads all replicas of every sector, to check for decay. If CAREFUL_GET reports that a replica of a sector is unreadable at one of these periodic checks, the clerk immediately rewrites that replica from a good one. If the rewrite fails, the clerk calls for immediate revectoring of that sector or, if the number of revectorings is rapidly growing, replacement of the decay set to which the sector belongs. The period between these checks should be short enough that the probability that *all* replicas have decayed since the previous check is negligible. By analyzing the statistics of experience for similar disk systems, the designer chooses such a period, T_d. This approach leads to the following failure tolerance model:

status ← MORE_DURABLE_PUT (*data*, *virtual_sector_number*)
status ← MORE_DURABLE_GET (*data*, *virtual_sector_number*)

- error-free operation: MORE_DURABLE_GET returns whatever was most recently written by MORE_DURABLE_PUT at *virtual_sector_number* with *status* = OK
- tolerated error: Hard errors reported by the careful storage layer are masked by reading from one of the other replicas. The result is that the operation completes with *status* = OK.
- tolerated error: data of a single decay set decays, is discovered by the clerk, and is repaired, all within T_d seconds of the decay event.
- untolerated error: The operating system crashes during a DURABLE_PUT and scribbles over the disk buffer in volatile storage, so DURABLE_PUT writes corrupted data on all mirror copies of that sector.
- untolerated error: all decay sets fail within T_d seconds. (With a conservative choice of T_d, the probability of this event should be negligible.)
- untolerated error: The data of some sector decays in a way that is undetectable—the checksum accidentally verifies. (With a good quality checksum, the probability of this event should be negligible.)

A somewhat less effective alternative to running a clerk that periodically verifies integrity of the data is to notice that the bathtub curve of Figure 8.1 applies to magnetic disks, and simply adopt a policy of systematically replacing the individual disks of the RAID array well before they reach the point where their conditional failure rate is predicted to start climbing. This alternative is not as effective for two reasons: First, it does not catch and repair random decay events, which instead accumulate. Second, it provides no warning if the actual operational lifetime is shorter than predicted (for example, if one happens to have acquired a bad batch of disks).

8.5.4.8 Detecting Errors Caused by System Crashes

With the addition of a clerk to watch for decay, there is now just one remaining untolerated error that has a significant probability: the hard error created by an operating system crash during CAREFUL_PUT. Since that scenario corrupts the data before the disk subsystem sees it, the disk subsystem has no way of either detecting or masking this error. Help is needed from outside the disk subsystem—either the operating system or the application. The usual approach is that either the system or, even better, the application program, calculates and includes an end-to-end checksum with the data before initiating the disk write. Any program that later reads the data verifies that the stored checksum matches the recalculated checksum of the data. The end-to-end checksum thus monitors the integrity of the data as it passes through the operating system buffers and also while it resides in the disk subsystem.

> **Sidebar 8.3: Are disk system checksums a wasted effort?** From the adjacent paragraph, an *end-to-end argument* suggests that an end-to-end checksum is always needed to protect data on its way to and from the disk subsystem, and that the fail-fast checksum performed inside the disk system thus may not be essential.
>
> However, the disk system checksum cleanly subcontracts one rather specialized job: correcting burst errors of the storage medium. In addition, the disk system checksum provides a handle for disk-layer erasure code implementations such as RAID, as was described in Section 8.4.1. Thus the disk system checksum, though superficially redundant, actually turns out to be quite useful.

The end-to-end checksum allows only detecting this class of error. Masking is another matter—it involves a technique called *recovery*, which is one of the topics of the next chapter.

Table 8.1 summarizes where failure tolerance is implemented in the several disk layers. The hope is that the remaining untolerated failures are so rare that they can be neglected. If they are not, the number of replicas could be increased until the probability of untolerated failures is negligible.

8.5.4.9 Still More Threats to Durability

The various procedures described above create storage that is durable in the face of individual disk decay but not in the face of other threats to data integrity. For example, if the power fails in the middle of a MORE_DURABLE_PUT, some replicas may contain old versions of the data, some may contain new versions, and some may contain corrupted data, so it is not at all obvious how MORE_DURABLE_GET should go about meeting its specification. The solution is to make MORE_DURABLE_PUT atomic, which is one of the topics of Chapter 9[on-line].

RAID systems usually specify that a successful return from a PUT confirms that writing of all of the mirror replicas was successful. That specification in turn usually requires that the multiple disks be physically co-located, which in turn creates a threat that a single physical disaster—fire, earthquake, flood, civil disturbance, etc.—might damage or destroy all of the replicas.

Since magnetic disks are quite reliable in the short term, a different strategy is to write only one replica at the time that MORE_DURABLE_PUT is invoked and write the remaining replicas at a later time. Assuming there are no inopportune failures in the short run, the results gradually become more durable as more replicas are written. Replica writes that are separated in time are less likely to have replicated failures because they can be separated in physical location, use different disk driver software, or be written to completely different media such as magnetic tape. On the other hand, separating replica writes in time increases the risk of inconsistency among the replicas. Implementing storage that has durability that is substantially beyond that of RAID 1 and MORE_DURABLE_PUT/GET generally involves use of geographically separated replicas and systematic mechanisms to keep those replicas coordinated, a challenge that Chapter 10[on-line] discusses in depth.

Perhaps the most serious threat to durability is that although different storage systems have employed each of the failure detection and masking techniques discussed in this section, it is all too common to discover that a typical off-the-shelf personal computer file

	raw layer	fail-fast layer	careful layer	durable layer	more durable layer
soft read, write, or seek error	failure	detected	masked		
hard read, write error	failure	detected	detected	masked	
power failure interrupts a write	failure	detected	detected	masked	
single data decay	failure	detected	detected	masked	
multiple data decay spaced in time	failure	detected	detected	detected	masked
multiple data decay within T_d	failure	detected	detected	detected	failure*
undetectable decay	failure	failure	failure	failure	failure*
system crash corrupts write buffer	failure	failure	failure	failure	detected

Table 8.1: Summary of disk failure tolerance models. Each entry shows the effect of this error at the interface between the named layer and the next higher layer. With careful design, the probability of the two failures marked with an asterisk should be negligible. Masking of corruption caused by system crashes is discussed in Chapter 9[on-line]

system has been designed using an overly simple disk failure model and thus misses some—or even many—straightforward failure masking opportunities.

8.6 Wrapping up Reliability

8.6.1 Design Strategies and Design Principles

Standing back from the maze of detail about redundancy, we can identify and abstract three particularly effective design strategies:

- *N-modular redundancy* is a simple but powerful tool for masking failures and increasing availability, and it can be used at any convenient level of granularity.
- *Fail-fast modules* provide a *sweeping simplification* of the problem of containing errors. When containment can be described simply, reasoning about fault tolerance becomes easier.
- *Pair-and-compare* allows fail-fast modules to be constructed from commercial, off-the-shelf components.

Standing back still further, it is apparent that several general design principles are directly applicable to fault tolerance. In the formulation of the fault-tolerance design process in Section 8.1.2, we invoked *be explicit*, *design for iteration*. *keep digging*, and the *safety margin principle*, and in exploring different fault tolerance techniques we have seen several examples of *adopt sweeping simplifications*. One additional design principle that applies to fault tolerance (and also, as we will see in Chapter 11[on-line], to security) comes from experience, as documented in the case studies of Section 8.8:

Avoid rarely used components

Deterioration and corruption accumulate unnoticed—until the next use.

Whereas redundancy can provide masking of errors, redundant components that are used only when failures occur are much more likely to cause trouble than redundant components that are regularly exercised in normal operation. The reason is that failures in regularly exercised components are likely to be immediately noticed and fixed. Failures in unused components may not be noticed until a failure somewhere else happens. But then there are two failures, which may violate the design assumptions of the masking plan. This observation is especially true for software, where rarely-used recovery procedures often accumulate unnoticed bugs and incompatibilities as other parts of the system evolve. The alternative of periodic testing of rarely-used components to lower their failure latency is a band-aid that rarely works well.

In applying these design principles, it is important to consider the threats, the consequences, the environment, and the application. Some faults are more likely than others, some failures are more disruptive than others, and different techniques may be appropriate in different environments. A computer-controlled radiation therapy machine, a deep-space probe, a telephone switch, and an airline reservation system all need fault tolerance, but in quite different forms. The radiation therapy machine should emphasize fault detection and fail-fast design, to avoid injuring patients. Masking faults may actually be a mistake. It is likely to be safer to stop, find their cause, and fix them before continuing operation. The deep-space probe, once the mission begins, needs to concentrate on failure masking to ensure mission success. The telephone switch needs many nines of availability because customers expect to always receive a dial tone, but if it occasionally disconnects one ongoing call, that customer will simply redial without thinking much about it. Users of the airline reservation system might tolerate short gaps in availability, but the durability of its storage system is vital. At the other extreme, most people find that a digital watch has an MTTF that is long compared with the time until the watch is misplaced, becomes obsolete, goes out of style, or is discarded. Consequently, no provision for either error masking or repair is really needed. Some applications have built-in redundancy that a designer can exploit. In a video stream, it is usually possible to mask the loss of a single video frame by just repeating the previous frame.

8.6.2 How about the End-to-End Argument?

There is a potential tension between error masking and an *end-to-end argument*. An end-to-end argument suggests that a subsystem need not do anything about errors and should not do anything that might compromise other goals such as low latency, high throughput, or low cost. The subsystem should instead let the higher layer system of which it is a component take care of the problem because only the higher layer knows whether or not the error matters and what is the best course of action to take.

There are two counter arguments to that line of reasoning:

- Ignoring an error allows it to propagate, thus contradicting the modularity goal of error containment. This observation points out an important distinction between error detection and error masking. Error detection and containment must be performed where the error happens, so that the error does not propagate wildly. Error masking, in contrast, presents a design choice: masking can be done locally or the error can be handled by reporting it at the interface (that is, by making the module design fail-fast) and allowing the next higher layer to decide what masking action—if any—to take.
- The lower layer may know the nature of the error well enough that it can mask it far more efficiently than the upper layer. The specialized burst error correction codes used on DVDs come to mind. They are designed specifically to mask errors caused by scratches and dust particles, rather than random bit-flips. So we have a trade-off between the cost of masking the fault locally and the cost of letting the error propagate and handling it in a higher layer.

These two points interact: When an error propagates it can contaminate otherwise correct data, which can increase the cost of masking and perhaps even render masking impossible. The result is that when the cost is small, error masking is usually done locally. (That is assuming that masking is done at all. Many personal computer designs omit memory error masking. Section 8.8.1 discusses some of the reasons for this design decision.)

A closely related observation is that when a lower layer masks a fault it is important that it also report the event to a higher layer, so that the higher layer can keep track of how much masking is going on and thus how much failure tolerance there remains. Reporting to a higher layer is a key aspect of *the safety margin principle*.

8.6.3 A Caution on the Use of Reliability Calculations

Reliability calculations seem to be exceptionally vulnerable to the garbage-in, garbage-out syndrome. It is all too common that calculations of mean time to failure are undermined because the probabilistic models are not supported by good statistics on the failure rate of the components, by measures of the actual load on the system or its components, or by accurate assessment of independence between components.

For computer systems, back-of-the-envelope calculations are often more than sufficient because they are usually at least as accurate as the available input data, which tends to be rendered obsolete by rapid technology change. Numbers predicted by formula can generate a false sense of confidence. This argument is much weaker for technologies that tend to be stable (for example, production lines that manufacture glass bottles). So reliability analysis is not a waste of time, but one must be cautious in applying its methods to computer systems.

8.6.4 Where to Learn More about Reliable Systems

Our treatment of fault tolerance has explored only the first layer of fundamental concepts. There is much more to the subject. For example, we have not considered another class of fault that combines the considerations of fault tolerance with those of security: faults caused by inconsistent, perhaps even malevolent, behavior. These faults have the characteristic they generate inconsistent error values, possibly error values that are specifically designed by an attacker to confuse or confound fault tolerance measures. These faults are called *Byzantine faults*, recalling the reputation of ancient Byzantium for malicious politics. Here is a typical Byzantine fault: suppose that an evil spirit occupies one of the three replicas of a TMR system, waits for one of the other replicas to fail, and then adjusts its own output to be identical to the incorrect output of the failed replica. A voter accepts this incorrect result and the error propagates beyond the intended containment boundary. In another kind of Byzantine fault, a faulty replica in an NMR system sends different result values to each of the voters that are monitoring its output. Malevolence is not required—any fault that is not anticipated by a fault detection mechanism can produce Byzantine behavior. There has recently been considerable attention to techniques that can tolerate Byzantine faults. Because the tolerance algorithms can be quite complex, we defer the topic to advanced study.

We also have not explored the full range of reliability techniques that one might encounter in practice. For an example that has not yet been mentioned, Sidebar 8.4 describes the *heartbeat*, a popular technique for detecting failures of active processes.

This chapter has oversimplified some ideas. For example, the definition of availability proposed in Section 8.2 of this chapter is too simple to adequately characterize many large systems. If a bank has hundreds of automatic teller machines, there will probably always be a few teller machines that are not working at any instant. For this case, an availability measure based on the percentage of transactions completed within a specified response time would probably be more appropriate.

A rapidly moving but in-depth discussion of fault tolerance can be found in Chapter 3 of the book *Transaction Processing: Concepts and Techniques*, by Jim Gray and Andreas Reuter. A broader treatment, with case studies, can be found in the book *Reliable Computer Systems: Design and Evaluation*, by Daniel P. Siewiorek and Robert S. Swarz. Byzantine faults are an area of ongoing research and development, and the best source is current professional literature.

This chapter has concentrated on general techniques for achieving reliability that are applicable to hardware, software, and complete systems. Looking ahead, Chapters 9[on-line] and 10[on-line] revisit reliability in the context of specific software techniques that permit reconstruction of stored state following a failure when there are several concurrent activities. Chapter 11[on-line], on securing systems against malicious attack, introduces a redundancy scheme known as *defense in depth* that can help both to contain and to mask errors in the design or implementation of individual security mechanisms.

Sidebar 8.4: Detecting failures with heartbeats. An activity such as a Web server is usually intended to keep running indefinitely. If it fails (perhaps by crashing) its clients may notice that it has stopped responding, but clients are not typically in a position to restart the server. Something more systematic is needed to detect the failure and initiate recovery. One helpful technique is to program the thread that should be performing the activity to send a periodic signal to another thread (or a message to a monitoring service) that says, in effect, "I'm still OK". The periodic signal is known as a *heartbeat* and the observing thread or service is known as a *watchdog*.

The watchdog service sets a timer, and on receipt of a heartbeat message it restarts the timer. If the timer ever expires, the watchdog assumes that the monitored service has gotten into trouble and it initiates recovery. One limitation of this technique is that if the monitored service fails in such a way that the only thing it does is send heartbeat signals, the failure will go undetected.

As with all fixed timers, choosing a good heartbeat interval is an engineering challenge. Setting the interval too short wastes resources sending and responding to heartbeat signals. Setting the interval too long delays detection of failures. Since detection is a prerequisite to repair, a long heartbeat interval increases MTTR and thus reduces availability.

8.7 Application: A Fault Tolerance Model for CMOS RAM

This section develops a fault tolerance model for words of CMOS random access memory, first without and then with a simple error-correction code, comparing the probability of error in the two cases.

CMOS RAM is both low in cost and extraordinarily reliable, so much so that error masking is often not implemented in mass production systems such as television sets and personal computers. But some systems, for example life-support, air traffic control, or banking systems, cannot afford to take unnecessary risks. Such systems usually employ the same low-cost memory technology but add incremental redundancy.

A common failure of CMOS RAM is that noise intermittently causes a single bit to read or write incorrectly. If intermittent noise affected only reads, then it might be sufficient to detect the error and retry the read. But the possibility of errors on writes suggests using a forward error-correction code.

We start with a fault tolerance model that applies when reading a word from memory without error correction. The model assumes that errors in different bits are independent and it assigns p as the (presumably small) probability that any individual bit is in error. The notation $O(p^n)$ means terms involving p^n and higher, presumably negligible, powers. Here are the possibilities and their associated probabilities:

Fault tolerance model for raw CMOS random access memory

		probability
error-free case:	all 32 bits are correct	$(1-p)^{32} = 1 - O(p)$
errors:		
untolerated:	one bit is in error:	$32p(1-p)^{31} = O(p)$
untolerated:	two bits are in error:	$(31 \cdot 32/2)p^2(1-p)^{30} = O(p^2)$
untolerated:	three or more bits are in error:	$(30 \cdot 31 \cdot 32/3 \cdot 2)p^3(1-p)^{29} + \ldots + p^{32} = O(p^3)$

The coefficients 32, $(31 \cdot 32)/2$, etc., arise by counting the number of ways that one, two, etc., bits could be in error.

Suppose now that the 32-bit block of memory is encoded using a code of Hamming distance 3, as described in Section 8.4.1. Such a code allows any single-bit error to be corrected and any double-bit error to be detected. After applying the decoding algorithm, the fault tolerance model changes to:

Fault tolerance model for CMOS memory with error correction

		probability
error-free case:	all 32 bits are correct	$(1-p)^{32} = 1 - O(p)$
errors:		
tolerated:	one bit corrected:	$32p(1-p)^{31} = O(p)$
detected:	two bits are in error:	$(31 \cdot 32/2)p^2(1-p)^{30} = O(p^2)$
untolerated:	three or more bits are in error:	$(30 \cdot 31 \cdot 32/3 \cdot 2)p^3(1-p)^{29} + \ldots + p^{32} = O(p^3)$

The interesting change is in the probability that the decoded value is correct. That probability is the sum of the probabilities that there were no errors and that there was one, tolerated error:

$$
\begin{aligned}
Prob(\text{decoded value is correct}) &= (1-p)^{32} + 32p(1-p)^{31} \\
&= (1 - 32p + (31 \cdot 32/2)p^2 + \ldots) + (32p + 31 \cdot \\
&= (1 - O(p^2))
\end{aligned}
$$

The decoding algorithm has thus eliminated the errors that have probability of order p. It has not eliminated the two-bit errors, which have probability of order $p2$, but for two-bit errors the algorithm is fail-fast, so a higher-level procedure has an opportunity to recover, perhaps by requesting retransmission of the data. The code is not helpful if there are errors in three or more bits, which situation has probability of order $p3$, but presumably the designer has determined that probabilities of that order are negligible. If they are not, the designer should adopt a more powerful error-correction code.

With this model in mind, one can review the two design questions suggested on page 8–19. The first question is whether the estimate of bit error probability is realistic and if it is realistic to suppose that multiple bit errors are statistically independent of one another. (Error independence appeared in the analysis in the claim that the probability of an n-bit error has the order of the nth power of the probability of a one-bit error.) Those questions concern the real world and the accuracy of the designer's model of it. For example, this failure model doesn't consider power failures, which might take all the bits out at once, or a driver logic error that might take out all of the even-numbered bits.

It also ignores the possibility of faults that lead to errors in the logic of the error-correction circuitry itself.

The second question is whether the coding algorithm actually corrects all one-bit errors and detects all two-bit errors. That question is explored by examining the mathematical structure of the error-correction code and is quite independent of anybody's estimate or measurement of real-world failure types and rates. There are many off-the-shelf coding algorithms that have been thoroughly analyzed and for which the answer is yes.

8.8 War Stories: Fault Tolerant Systems that Failed

8.8.1 Adventures with Error Correction*

The designers of the computer systems at the Xerox Palo Alto Research Center in the early 1970s encountered a series of experiences with error-detecting and error-correcting memory systems. From these experiences follow several lessons, some of which are far from intuitive, and all of which still apply several decades later.

MAXC. One of the first projects undertaken in the newly-created Computer Systems Laboratory was to build a time-sharing computer system, named MAXC. A brand new 1024-bit memory chip, the Intel 1103, had just appeared on the market, and it promised to be a compact and economical choice for the main memory of the computer. But since the new chip had unknown reliability characteristics, the MAXC designers implemented the memory system using a few extra bits for each 36-bit word, in the form of a single-error-correction, double-error-detection code.

Experience with the memory in MAXC was favorable. The memory was solidly reliable—so solid that no errors in the memory system were ever reported.

The Alto. When the time came to design the Alto personal workstation, the same Intel memory chips still appeared to be the preferred component. Because these chips had performed so reliably in MAXC, the designers of the Alto memory decided to relax a little, omitting error correction. But, they were still conservative enough to provide error detection, in the form of one parity bit for each 16-bit word of memory.

This design choice seemed to be a winner because the Alto memory systems also performed flawlessly, at least for the first several months. Then, mysteriously, the operating system began to report frequent memory-parity failures.

Some background: the Alto started life with an operating system and applications that used a simple typewriter-style interface. The display was managed with a character-by-character teletype emulator. But the purpose of the Alto was to experiment with better

* These experiences were reported by Butler Lampson, one of the designers of the MAXC computer and the Alto personal workstations at Xerox Palo Alto Research Center.

things. One of the first steps in that direction was to implement the first what-you-see-is-what-you-get editor, named Bravo. Bravo took full advantage of the bit-map display, filling it not only with text, but also with lines, buttons, and icons. About half the memory system was devoted to display memory. Curiously, the installation of Bravo coincided with the onset of memory parity errors.

It turned out that the Intel 1103 chips were pattern-sensitive—certain read/write sequences of particular bit patterns could cause trouble, probably because those pattern sequences created noise levels somewhere on the chip that systematically exceeded some critical threshold. The Bravo editor's display management was the first application that generated enough different patterns to have an appreciable probability of causing a parity error. It did so, frequently.

Lesson 8.8.1a: There is no such thing as a small change in a large system. A new piece of software can bring down a piece of hardware that is thought to be working perfectly. You are never quite sure just how close to the edge of the cliff you are standing.

Lesson 8.8.1b: Experience is a primary source of information about failures. It is nearly impossible, without specific prior experience, to predict what kinds of failures you will encounter in the field.

Back to MAXC. This circumstance led to a more careful review of the situation on MAXC. MAXC, being a heavily used server, would be expected to encounter at least some of this pattern sensitivity. It was discovered that although the error-correction circuits had been designed to report both corrected errors and uncorrectable errors, the software logged only uncorrectable errors and corrected errors were being ignored. When logging of corrected errors was implemented, it turned out that the MAXC's Intel 1103's were actually failing occasionally, and the error-correction circuitry was busily setting things right.

Lesson 8.8.1c: Whenever systems implement automatic error masking, it is important to follow the safety margin principle, by tracking how often errors are successfully masked. Without this information, one has no way of knowing whether the system is operating with a large or small safety margin for additional errors. Otherwise, despite the attempt to put some guaranteed space between yourself and the edge of the cliff, you may be standing on the edge again.

The Alto 2. In 1975, it was time to design a follow-on workstation, the Alto 2. A new generation of memory chips, this time with 4096 bits, was now available. Since it took up much less space and promised to be cheaper, this new chip looked attractive, but again there was no experience with its reliability. The Alto 2 designers, having been made wary by the pattern sensitivity of the previous generation chips, again resorted to a single-error-correction, double-error-detection code in the memory system.

Once again, the memory system performed flawlessly. The cards passed their acceptance tests and went into service. In service, not only were no double-bit errors detected, only rarely were single-bit errors being corrected. The initial conclusion was that the chip vendors had worked the bugs out and these chips were really good.

About two years later, someone discovered an implementation mistake. In one quadrant of each memory card, neither error correction nor error detection was actually working. All computations done using memory in the misimplemented quadrant were completely unprotected from memory errors.

Lesson 8.8.1d: Never assume that the hardware actually does what it says in the specifications.
Lesson 8.8.1e: It is harder than it looks to test the fault tolerance features of a fault tolerant system.

One might conclude that the intrinsic memory chip reliability had improved substantially—so much that it was no longer necessary to take heroic measures to achieve system reliability. Certainly the chips were better, but they weren't perfect. The other effect here is that errors often don't lead to failures. In particular, a wrong bit retrieved from memory does not necessarily lead to an observed failure. In many cases a wrong bit doesn't matter; in other cases it does but no one notices; in still other cases, the failure is blamed on something else.

Lesson 8.8.1f: Just because it seems to be working doesn't mean that it actually is.

The bottom line. One of the designers of MAXC and the Altos, Butler Lampson, suggests that the possibility that a failure is blamed on something else can be viewed as an opportunity, and it may be one of the reasons that PC manufacturers often do not provide memory parity checking hardware. First, the chips are good enough that errors are rare. Second, if you provide parity checks, consider who will be blamed when the parity circuits report trouble: the hardware vendor. Omitting the parity checks probably leads to occasional random behavior, but occasional random behavior is indistinguishable from software error and is usually blamed on the software.

Lesson 8.8.1g (in Lampson's words): "Beauty is in the eye of the beholder. The various parties involved in the decisions about how much failure detection and recovery to implement do not always have the same interests."

8.8.2 Risks of Rarely-Used Procedures: The National Archives

The National Archives and Record Administration of the United States government has the responsibility, among other things, of advising the rest of the government how to preserve electronic records such as e-mail messages for posterity. Quite separate from that responsibility, the organization also operates an e-mail system at its Washington, D.C. headquarters for a staff of about 125 people and about 10,000 messages a month pass through this system. To ensure that no messages are lost, it arranged with an outside contractor to perform daily incremental backups and to make periodic complete backups of its e-mail files. On the chance that something may go wrong, the system has audit logs that track actions regarding incoming and outgoing mail as well as maintenance on files.

Over the weekend of June 18–21, 1999, the e-mail records for the previous four months (an estimated 43,000 messages) disappeared. No one has any idea what went wrong—the files may have been deleted by a disgruntled employee or a runaway housecleaning program, or the loss may have been caused by a wayward system bug. In any case, on Monday morning when people came to work, they found that the files were missing.

On investigation, the system managers reported that the audit logs had been turned off because they were reducing system performance, so there were no clues available to diagnose what went wrong. Moreover, since the contractor's employees had never gotten around to actually performing the backup part of the contract, there were no backup copies. It had not occurred to the staff of the Archives to verify the existence of the backup copies, much less to test them to see if they could actually be restored. They assumed that since the contract required it, the work was being done.

The contractor's project manager and the employee responsible for making backups were immediately replaced. The Assistant Archivist reports that backup systems have now been beefed up to guard against another mishap, but he added that the safest way to save important messages is to print them out.*

Lesson 8.8.2: Avoid rarely used components. *Rarely used failure-tolerance mechanisms, such as restoration from backup copies, must be tested periodically. If they are not, there is not much chance that they will work when an emergency arises. Fire drills (in this case performing a restoration of all files from a backup copy) seem disruptive and expensive, but they are not nearly as disruptive and expensive as the discovery, too late, that the backup system isn't really operating. Even better, design the system so that all the components are exposed to day-to-day use, so that failures can be noticed before they cause real trouble.*

8.8.3 Non-independent Replicas and Backhoe Fade

In Eagan, Minnesota, Northwest airlines operated a computer system, named WorldFlight, that managed the Northwest flight dispatching database, provided weight-and-balance calculations for pilots, and managed e-mail communications between the dispatch center and all Northwest airplanes. It also provided data to other systems that managed passenger check-in and the airline's Web site. Since many of these functions involved communications, Northwest contracted with U.S. West, the local telephone company at that time, to provide these communications in the form of fiber-optic links to airports that Northwest serves, to government agencies such as the Weather Bureau and the Federal Aviation Administration, and to the Internet. Because these links were vital, Northwest paid U.S. West extra to provide each primary link with a backup secondary link. If a primary link to a site failed, the network control computers automatically switched over to the secondary link to that site.

At 2:05 p.m. on March 23, 2000, all communications to and from WorldFlight dropped out simultaneously. A contractor who was boring a tunnel (for fiber optic lines for a different telephone company) at the nearby intersection of Lone Oak and Pilot Knob roads accidentally bored through a conduit containing six cables carrying the U.S.

* George Lardner Jr. "Archives Loses 43,000 E-Mails; officials can't explain summer erasure; backup system failed." *The Washington Post,* Thursday, January 6, 2000, page A17.

West fiber-optic and copper lines. In a tongue-in-cheek analogy to the fading in and out of long-distance radio signals, this kind of communications disruption is known in the trade as "backhoe fade." WorldFlight immediately switched from the primary links to the secondary links, only to find that they were not working, either. It seems that the primary and secondary links were routed through the same conduit, and both were severed.

Pilots resorted to manual procedures for calculating weight and balance, and radio links were used by flight dispatchers in place of the electronic message system, but about 125 of Northwest's 1700 flights had to be cancelled because of the disruption, about the same number that are cancelled when a major snowstorm hits one of Northwest's hubs. Much of the ensuing media coverage concentrated on whether or not the contractor had followed "dig-safe" procedures that are intended to prevent such mistakes. But a news release from Northwest at 5:15 p.m. blamed the problem entirely on U.S. West. "For such contingencies, U.S. West provides to Northwest a complete redundancy plan. The U.S. West redundancy plan also failed."*

In a similar incident, the ARPAnet, a predecessor to the Internet, had seven separate trunk lines connecting routers in New England to routers elsewhere in the United States. All the trunk lines were purchased from a single long-distance carrier, AT&T. On December 12, 1986, all seven trunk lines went down simultaneously when a contractor accidentally severed a single fiber-optic cable running from White Plains, New York to Newark, New Jersey.†

A complication for communications customers who recognize this problem and request information about the physical location of their communication links is that, in the name of security, communications companies sometimes refuse to reveal it.

Lesson 8.8.3: The calculation of mean time to failure of a redundant system depends critically on the assumption that failures of the replicas are independent. If they aren't independent, then the replication may be a waste of effort and money, while producing a false complacency. This incident also illustrates why it can be difficult to test fault tolerance measures properly. What appears to be redundancy at one level of abstraction turns out not to be redundant at a lower level of abstraction.

8.8.4 Human Error May Be the Biggest Risk

Telehouse was an East London "telecommunications hotel", a seven story building housing communications equipment for about 100 customers, including most British Internet companies, many British and international telephone companies, and dozens of financial institutions. It was designed to be one of the most secure buildings in Europe, safe against "fire, flooding, bombs, and sabotage". Accordingly, Telehouse had extensive protection against power failure, including two independent connections to the national

* Tony Kennedy. "Cut cable causes cancellations, delays for Northwest Airlines." *Minneapolis Star Tribune*, March 22, 2000.

† Peter G. Neumann. *Computer Related Risks* (Addison-Wesley, New York, 1995), page 14.

electric power grid, a room full of batteries, and two diesel generators, along with systems to detect failures in supply and automatically cut over from one backup system to the next, as needed.

On May 8, 1997, all the computer systems went off line for lack of power. According to Robert Bannington, financial director of Telehouse, “It was due to human error.” That is, someone pulled the wrong switch. The automatic power supply cutover procedures did not trigger because they were designed to deploy on failure of the outside power supply, and the sensors correctly observed that the outside power supply was intact.*

Lesson 8.8.4a: The first step in designing a fault tolerant system is to identify each potential fault and evaluate the risk that it will happen. People are part of the system, and mistakes made by authorized operators are typically a bigger threat to reliability than trees falling on power lines.

Anecdotes concerning failures of backup power supply systems seem to be common. Here is a typical report of an experience in a Newark, New Jersey, hospital operating room that was equipped with three backup generators: “On August 14, 2003, at 4:10pm EST, a widespread power grid failure caused our hospital to suffer a total OR power loss, regaining partial power in 4 hours and total restoration 12 hours later... When the backup generators initially came on-line, all ORs were running as usual. Within 20 minutes, one parallel-linked generator caught fire from an oil leak. After being subjected to twice its rated load, the second in-line generator quickly shut down... Hospital engineering, attempting load-reduction to the single surviving generator, switched many hospital circuit breakers off. Main power was interrupted to the OR.”†

Lesson 8.8.4b: A backup generator is another example of a rarely used component that may not have been maintained properly. The last two sentences of that report reemphasize Lesson 8.8.4a.

For yet another example, the M.I.T. Information Services and Technology staff posted the following system services notice on April 2, 2004: “We suffered a power failure in W92 shortly before 11AM this morning. Most services should be restored now, but some are still being recovered. Please check back here for more information as it becomes available.” A later posting reported: “Shortly after 10AM Friday morning the routine test of the W92 backup generator was started. Unknown to us was that the transition of the computer room load from commercial power to the backup generator resulted in a power surge within the computer room's Uninterruptable [sic] Power Supply (UPS). This destroyed an internal surge protector, which started to smolder. Shortly before 11AM the smoldering protector triggered the VESDA® smoke sensing system

* Robert Uhlig. “Engineer pulls plug on secure bunker.” *Electronic Telegraph,* (9 May 1997).

† Ian E. Kirk, M.D. and Peter L. Fine, M.D. “Operating by Flashlight: Power Failure and Safety Lessons from the August, 2003 Blackout.” *Abstracts of the Annual Meeting of the American Society of Anesthesiologists*, October 2005.

within the computer room. This sensor triggered the fire alarm, and as a safety precaution forced an emergency power down of the entire computer room."*

Lesson 8.8.4c: A failure masking system not only can fail, it can cause a bigger failure than the one it is intended to mask.

8.8.5 Introducing a Single Point of Failure

"[Rabbi Israel Meir HaCohen Kagan described] a real-life situation in his town of Radin, Poland. He lived at the time when the town first purchased an electrical generator and wired all the houses and courtyards with electric lighting. One evening something broke within the machine, and darkness descended upon all of the houses and streets, and even in the synagogue.

"So he pointed out that before they had electricity, every house had a kerosene light—and if in one particular house the kerosene ran out, or the wick burnt away, or the glass broke, that only that one house would be dark. But when everyone is dependent upon one machine, darkness spreads over the entire city if it breaks for any reason."†

Lesson 8.8.5: Centralization may provide economies of scale, but it can also reduce robustness—a single failure can interfere with many unrelated activities. This phenomenon is commonly known as introducing a single point of failure. By carefully adding redundancy to a centralized design one may be able to restore some of the lost robustness but it takes planning and adds to the cost.

8.8.6 Multiple Failures: The SOHO Mission Interruption

"Contact with the SOlar Heliospheric Observatory (SOHO) spacecraft was lost in the early morning hours of June 25, 1998, Eastern Daylight Time (EDT), during a planned period of calibrations, maneuvers, and spacecraft reconfigurations. Prior to this the SOHO operations team had concluded two years of extremely successful science operations.

"…The Board finds that the loss of the SOHO spacecraft was a direct result of operational errors, a failure to adequately monitor spacecraft status, and an erroneous decision which disabled part of the on-board autonomous failure detection. Further, following the occurrence of the emergency situation, the Board finds that insufficient time was taken by the operations team to fully assess the spacecraft status prior to initiating recovery operations. The Board discovered that a number of factors contributed to the circumstances that allowed the direct causes to occur."‡

---

* Private internal communication.

† Chofetz Chaim (the Rabbi Israel Meir HaCohen Kagan of Radin), paraphrased by Rabbi Yaakov Menken, in a discussion of lessons from the Torah in *Project Genesis Lifeline*. `<http://www.torah.org/learning/lifeline/5758/reeh.html>`. Suggested by David Karger.

In a tour-de-force of the *keep digging principle*, the report of the investigating board quoted above identified five distinct direct causes of the loss: two software errors, a design feature that unintentionally amplified the effect of one of the software errors, an incorrect diagnosis by the ground staff, and a violated design assumption. It then goes on to identify three indirect causes in the spacecraft design process: lack of change control, missing risk analysis for changes, and insufficient communication of changes, and then three indirect causes in operations procedures: failure to follow planned procedures, to evaluate secondary telemetry data, and to question telemetry discrepancies.

Lesson 8.8.6: Complex systems fail for complex reasons. In systems engineered for reliability, it usually takes several component failures to cause a system failure. Unfortunately, when some of the components are people, multiple failures are all too common.

Exercises

8.1 Failures are

A. Faults that are latent.
B. Errors that are contained within a module.
C. Errors that propagate out of a module.
D. Faults that turn into errors.

1999–3–01

8.2 Ben Bitdiddle has been asked to perform a deterministic computation to calculate the orbit of a near-Earth asteroid for the next 500 years, to find out whether or not the asteroid will hit the Earth. The calculation will take roughly two years to complete, and Ben wants be be sure that the result will be correct. He buys 30 identical computers and runs the same program with the same inputs on all of them. Once each hour the software pauses long enough to write all intermediate results to a hard disk on that computer. When the computers return their results at the end

‡ Massimo Trella and Michael Greenfield. *Final Report of the SOHO Mission Interruption Joint NASA/ESA Investigation Board* (August 31, 1998). National Aeronautics and Space Administration and European Space Agency.
<http://sohowww.nascom.nasa.gov/whatsnew/SOHO_final_report.html>

of the two years, a voter selects the majority answer. Which of the following failures can this scheme tolerate, assuming the voter works correctly?

A. The software carrying out the deterministic computation has a bug in it, causing the program to compute the wrong answer for certain inputs.
B. Over the course of the two years, cosmic rays corrupt data stored in memory at twelve of the computers, causing them to return incorrect results.
C. Over the course of the two years, on 24 different days the power fails in the computer room. When the power comes back on, each computer reboots and then continues its computation, starting with the state it finds on its hard disk.

2006–2–3

8.3 Ben Bitdiddle has seven smoke detectors installed in various places in his house. Since the fire department charges $100 for responding to a false alarm, Ben has connected the outputs of the smoke detectors to a simple majority voter, which in turn can activate an automatic dialer that calls the fire department. Ben returns home one day to find his house on fire, and the fire department has not been called. There is smoke at every smoke detector. What did Ben do wrong?

A. He should have used fail-fast smoke detectors.
B. He should have used a voter that ignores failed inputs from fail-fast sources.
C. He should have used a voter that ignores non-active inputs.
D. He should have done both A and B.
E. He should have done both Aand C.

1997–0–01

8.4 You will be flying home from a job interview in Silicon Valley. Your travel agent gives you the following choice of flights:

A. Flight A uses a plane whose mean time to failure (MTTF) is believed to be 6,000 hours. With this plane, the flight is scheduled to take 6 hours.
B. Flight B uses a plane whose MTTF is believed to be 5,000 hours. With this plane, the flight takes 5 hours.

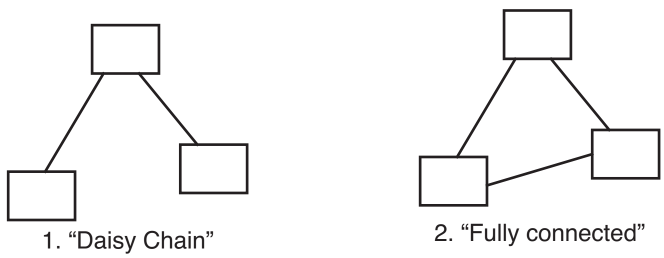
The agent assures you that both planes' failures occur according to memoryless random processes (not a "bathtub" curve). Assuming that model, which flight should you choose to minimize the chance of your plane failing during the flight?

2005–2–5

8.5 (Note: solving this problem is best done with use of probability through the level of Markov chains.) You are designing a computer system to control the power grid for the Northeastern United States. If your system goes down, the lights go out and civil disorder—riots, looting, fires, etc.—will ensue. Thus, you have set a goal of having a system MTTF of at least 100 years (about 10^6 hours). For hardware you are constrained to use a building block computer that has a MTTF of 1000 hours and a MTTR of 1 hour. Assuming that the building blocks are fail-fast, memoryless, and fail independently of one another, how can you arrange to meet your goal?

1995–3–1a

8.6 The town council wants to implement a municipal network to connect the local area networks in the library, the town hall, and the school. They want to minimize the chance that any building is completely disconnected from the others. They are considering two network topologies:



Each link in the network has a failure probability of p.

8.6a. What is the probability that the daisy chain network is connecting all the buildings?

8.6b. What is the probability that the fully connected network is connecting all the buildings?

8.6c. The town council has a limited budget, with which it can buy either a daisy chain network with two high reliability links ($p = .000001$), or a fully connected network with three low-reliability links ($p = .0001$). Which should they purchase?

1985–0–1

8.7 Figure 8.11 shows the failure points of three different 5MR supermodule designs, if repair does not happen in time. Draw the corresponding figure for the same three different TMR supermodule designs.

2001–3–05

8.8 An astronomer calculating the trajectory of Pluto has a program that requires the execution of 10^{13} machine operations. The fastest processor available in the lab runs only 10^{9} operations per second and, unfortunately, has a probability of failing on any one operation of 10^{-12}. (The failure process is memoryless.) The good news is that the processor is fail-fast, so when a failure occurs it stops dead in its tracks and starts ringing a bell. The bad news is that when it fails, it loses all state, so whatever it was doing is lost, and has to be started over from the beginning.

Seeing that in practical terms, the program needs to run for about 3 hours, and the machine has an MTTF of only 1/10 of that time, Louis Reasoner and Ben Bitdiddle have proposed two ways to organize the computation:

- Louis says run it from the beginning and hope for the best. If the machine fails, just try again; keep trying till the calculation successfully completes.
- Ben suggests dividing the calculation into ten equal-length segments; if the calculation gets to the end of a segment, it writes its state out to the disk. When a failure occurs, restart from the last state saved on the disk.

Saving state and restart both take zero time. What is the ratio of the expected time to complete the calculation under the two strategies?

Warning: A straightforward solution to this problem involves advanced probability techniques.

1976–0–3

8.9 Draw a figure, similar to that of Figure 8.6, that shows the recovery procedure for one sector of a 5-disk RAID 4 system when disk 2 fails and is replaced.

2005–0–1

8.10 Louis Reasoner has just read an advertisement for a RAID controller that provides a choice of two configurations. According to the advertisement, the first configuration is exactly the RAID 4 system described in Section 8.4.1. The advertisement goes on to say that the configuration called RAID 5 has just one difference: in an *N*-disk configuration, the parity block, rather than being written on disk *N*, is written on the disk number (1 + *sector_address* modulo *N*). Thus, for example, in a five-disk system, the parity block for sector 18 would be on disk 4 (because 1 + (18 modulo 5) = 4), while the parity block for sector 19 would be on disk 5 (because 1 + (19 modulo 5) = 5). Louis is hoping you can help him understand why this idea might be a good one.

8.10a. RAID 5 has the advantage over RAID 4 that

A. It tolerates single-drive failures.
B. Read performance in the absence of errors is enhanced.
C. Write performance in the absence of errors is enhanced.
D. Locating data on the drives is easier.
E. Allocating space on the drives is easier.
F. It requires less disk space.
G. There's no real advantage, its just another advertising gimmick.

1997–3–01

8.10b. Is there any workload for which RAID 4 has better write performance than RAID 5?

2000–3–01

8.10c. Louis is also wondering about whether he might be better off using a RAID 1 system (see Section 8.5.4.6). How does the number of disks required compare between RAID 1 and RAID 5?

1998–3–01

8.10d. Which of RAID 1 and RAID 5 has better performance for a workload consisting of small reads and small writes?

2000–3–01

8.11 A system administrator notices that a file service disk is failing for two unrelated reasons. Once every 30 days, on average, vibration due to nearby construction breaks the disk's arm. Once every 60 days, on average, a power surge destroys the disk's electronics. The system administrator fixes the disk instantly each time it fails. The two failure modes are independent of each other, and independent of the age of the disk. What is the mean time to failure of the disk?

2002–3–01

Additional exercises relating to Chapter 8 can be found in problem sets *26* through *28*.

Glossary for Chapter 8

active fault—A fault that is currently causing an error. Compare with *latent fault*. [Ch. 8]

availability—A measure of the time that a system was actually usable, as a fraction of the time that it was intended to be usable. Compare with its complement, *down time*. [Ch. 8]

backward error correction—A technique for correcting errors in which the source of the data or control signal applies enough redundancy to allow errors to be detected and, if an error does occur, that source is asked to redo the calculation or repeat the transmission. Compare with *forward error correction*. [Ch. 8]

Byzantine fault—A fault that generates inconsistent errors (perhaps maliciously) that can confuse or disrupt fault tolerance or security mechanisms. [Ch. 8]

close-to-open consistency—A consistency model for file operations. When a thread opens a file and performs several write operations, all of the modifications weill be visible to concurrent threads only after the first thread closes the file. [Ch. 4]

coheerence—See *read/write coherence* or *cache coherence*.

continuous operation—An availability goal, that a system be capable of running indefinitely. The primary requirement of continuous operation is that it must be possible to perform repair and maintenance without stopping the system. [Ch. 8]

decay set—A set of storage blocks, words, tracks, or other physical groupings, in which all members of the set may spontaneously fail together, but independently of any other decay set. [Ch. 8]

detectable error—An error or class of errors for which a reliable detection plan can be devised. An error that is not detectable usually leads to a failure, unless some mechanism that is intended to mask some other error accidentally happens to mask the undetectable error. Compare with *maskable error* and *tolerated error*. [Ch. 8]

down time—A measure of the time that a system was not usable, as a fraction of the time that it was intended to be usable. Compare with its complement, *availability*. [Ch. 8]

durable storage—Storage with the property that it (ideally) is decay-free, so it never fails to return on a GET the data that was stored by a previously successful PUT. Since that ideal is impossibly strict, in practice, storage is considered durable when the probability of failure is sufficiently low that the application can tolerate it. Durability is thus an application-defined specification of how long the results of an action, once completed, must be preserved. Durable is distinct from *non-volatile*, which describes storage that maintains its memory while the power is off, but may still have an intolerable probability of decay. The term *persistent* is sometimes used as a synonym for durable, as explained in Sidebar 2.7, but to minimize confusion this text avoids that usage. [Ch. 8]

erasure—An error in a string of bits, bytes, or groups of bits in which an identified bit, byte, or group of bits is missing or has indeterminate value. [Ch. 8]

ergodic—A property of some time-dependent probabilistic processes: that the (usually easier to measure) ensemble average of some parameter measured over a set of elements subject to the process is the same as the time average of that parameter of any single element of the ensemble. [Ch. 8]

error—Informally, a label for an incorrect data value or control signal caused by an active fault. If there is a complete formal specification for the internal design of a module, an error is a violation of some assertion or invariant of the specification. An error in a module is not identical to a failure of that module, but if an error is not masked, it may lead to a failure of the module. [Ch. 8]

error containment—Limiting how far the effects of an error propagate. A module is normally designed to contain errors in such a way that the effects of an error appear in a predictable way at the module's interface. [Ch. 8]

error correction—A scheme to set to the correct value a data value or control signal that is in error. Compare with *error detection.* [Ch. 8]

error detection—A scheme to discover that a data value or control signal is in error. Compare with *error correction.* [Ch. 8]

fail-fast—Describes a system or module design that contains detected errors by reporting at its interface that its output may be incorrect. Compare with *fail-stop*. [Ch. 8]

fail-safe—Describes a system design that detects incorrect data values or control signals and forces them to values that, even if not correct, are known to allow the system to continue operating safely. [Ch. 8]

fail-secure—Describes an application of fail-safe design to information protection: a failure is guaranteed not to allow unauthorized access to protected information. In early work on fault tolerance, this term was also occasionally used as a synonym for *fail-fast*. [Ch. 8]

fail-soft—Describes a design in which the system specification allows errors to be masked by degrading performance or disabling some functions in a predictable manner. [Ch. 8]

fail-stop—Describes a system or module design that contains detected errors by stopping the system or module as soon as possible. Compare with *fail-fast*, which does not require other modules to take additional action, such as setting a timer, to detect the failure. [Ch. 8]

fail-vote—Describes an *N*-modular redundancy system with a majority voter. [Ch. 8]

failure—The outcome when a component or system does not produce the intended result at its interface. Compare with **fault**. [Ch. 8]

failure tolerance—A measure of the ability of a system to mask active faults and continue operating correctly. A typical measure counts the number of contained components that can fail without causing the system to fail. [Ch. 8]

fault—A defect in materials, design, or implementation that may (or may not) cause an error and lead to a failure. (Compare with *failure*.) [Ch. 8]

fault avoidance—A strategy to design and implement a component with a probability of faults that is so low that it can be neglected. When applied to software, fault avoidance is sometimes called *valid construction*. [Ch. 8]

fault tolerance—A set of techniques that involve noticing active faults and lower-level subsystem failures and masking them, rather than allowing the resulting errors to propagate. [Ch. 8]

forward error correction—A technique for controlling errors in which enough redundancy to correct anticipated errors is applied before an error occurs. Forward error correction is particularly applicable when the original source of the data value or control signal will not be available to recalculate or resend it. Compare with *backward error correction*. [Ch. 8]

Hamming distance—in an encoding system, the number of bits in an element of a code that would have to change to transform it into a different element of the code. The Hamming distance of a code is the minimum Hamming distance between any pair of elements of the code. [Ch. 8]

hot swap—To replace modules in a system while the system continues to provide service. [Ch. 8]

intermittent fault—A persistent fault that is active only occasionally. Compare with *transient fault*. [Ch. 8]

latency—As used in reliability, the time between when a fault becomes active and when the module in which the fault occurred either fails or detects the resulting error. [Ch. 8]

latent fault—A fault that is not currently causing an error. Compare with *active fault*. [Ch. 8]

margin—The amount by which a specification is better than necessary for correct operation. The purpose of designing with margins is to mask some errors. [Ch. 8]

maskable error—An error or class of errors that is detectable and for which a systematic recovery strategy can in principle be devised. Compare with *detectable error* and *tolerated error*. [Ch. 8]

masking—As used in reliability, containing an error within a module in such a way that the module meets its specifications as if the error had not occurred. [Ch. 8]

mean time between failures (MTBF)—The sum of MTTF and MTTR for the same component or system. [Ch. 8]

mean time to failure (MTTF)—The expected time that a component or system will operate continuously without failing. “Time” is sometimes measured in cycles of operation. [Ch. 8]

mean time to repair (MTTR)—The expected time to replace or repair a component or system that has failed. The term is sometimes written as “mean time to restore service”, but it is still abbreviated MTTR. [Ch. 8]

memoryless—A property of some time-dependent probabilistic processes, that the probability of what happens next does not depend on what has happened before. [Ch. 8]

mirror—(n.) One of a set of replicas that is created or updated synchronously. Compare with *primary copy* and *backup copy*. Sometimes used as a verb, as in "Let's mirror that data by making three replicas." [Ch. 8]

N + 1 redundancy—When a load can be handled by sharing it among N equivalent modules, the technique of installing $N + 1$ or more of the modules, so that if one fails the remaining modules can continue to handle the full load while the one that failed is being repaired. [Ch. 8]

N-modular redundancy (NMR)—A redundancy technique that involves supplying identical inputs to N equivalent modules and connecting the outputs to one or more voters. [Ch. 8]

N-version programming—The software version of N-modular redundancy. N different teams each independently write a program from its specifications. The programs then run in parallel, and voters compare their outputs. [Ch. 8]

page fault—See *missing-page exception*.

pair-and-compare—A method for constructing fail-fast modules from modules that do not have that property, by connecting the inputs of two replicas of the module together and connecting their outputs to a comparator. When one repairs a failed pair-and-compare module by replacing the entire two-replica module with a spare, rather than identifying and replacing the replica that failed, the method is called **pair-and-spare.** [Ch. 8]

pair-and-spare—See *pair-and-compare*.

partition—To divide a job up and assign it to different physical devices, with the intent that a failure of one device does not prevent the entire job from being done. [Ch. 8]

persistent fault—A fault that cannot be masked by retry. Compare with *transient fault* and *intermittent fault*. [Ch. 8]

prepaging—An optimization for a multilevel memory manager in which the manager predicts which pages might be needed and brings them into the primary memory before the application demands them. Compare with *demand algorithm*.

presented load—See *offered load*.

preventive maintenance—Active intervention intended to increase the mean time to failure of a module or system and thus improve its reliability and availability. [Ch. 8]

purging—A technique used in some N-modular redundancy designs, in which the voter ignores the output of any replica that, at some time in the past, disagreed with several others. [Ch. 8]

redundancy—Extra information added to detect or correct errors in data or control signals. [Ch. 8]

reliability—A statistical measure, the probability that a system is still operating at time t, given that it was operating at some earlier time t_0. [Ch. 8]

repair—An active intervention to fix or replace a module that has been identified as failing, preferably before the system of which it is a part fails. [Ch. 8]

replica—1. One of several identical modules that, when presented with the same inputs, is expected to produce the same output. 2. One of several identical copies of a set of data. [Ch. 8]

replication—The technique of using multiple replicas to achieve fault tolerance. [Ch. 8]

single-event upset—A synonym for *transient fault*. [Ch. 8]

soft state—State of a running program that the program can easily reconstruct if it becomes necessary to abruptly terminate and restart the program. [Ch. 8]

supermodule—A set of replicated modules interconnected in such a way that it acts like a single module. [Ch. 8]

tolerated error—An error or class of errors that is both detectable and maskable, and for which a systematic recovery procedure has been implemented. Compare with *detectable error*, *maskable error*, and *untolerated error*. [Ch. 8]

transient fault—A fault that is temporary and for which retry of the putatively failed component has a high probability of finding that it is okay. Sometimes called a *single-event upset*. Compare with *persistent fault* and **intermittent fault**. [Ch. 8]

triple-modular redundancy (TMR)—N-modular redundancy with $N = 3$. [Ch. 8]

untolerated error—An error or class of errors that is undetectable, unmaskable, or unmasked and therefore can be expected to lead to a failure. Compare with *detectable error*, *maskable error*, and *tolerated error*. [Ch. 8]

valid construction—The term used by software designers for *fault avoidance*. [Ch. 8]

voter—A device used in some NMR designs to compare the output of several nominally identical replicas that all have the same input. [Ch. 8]

Index of Chapter 8

Design principles and hints appear in *underlined italics*. Procedure names appear in SMALL CAPS. Page numbers in **bold face** are in the chapter Glossary.

Austin Group Defect Tracker

Aardvark Mark III

Anonymous | Login | Signup for a new account   2015-03-04 04:25 UTC   Project: All Projects Switch RSS

Main | My View | View Issues | Change Log | Docs   Issue # Jump

Viewing Issue Simple Details [Jump to Notes]   [Issue History] [Print]

ID	Category	Severity	Type	Date Submitted	Last Update
0000672	[1003.1(2008)/Issue 7] System Interfaces	Editorial	Omission	2013-03-19 15:45	2013-08-01 10:25

Reporter	xroche	**View Status**	public
Assigned To	ajosey		
Priority	normal	**Resolution**	Open
Status	Under Review		
Name	Xavier Roche		
Organization			
User Reference			
Section			
Page Number			
Line Number			
Interp Status	---		
Final Accepted Text			

Summary 0000672: Necessary step(s) to synchronize filename operations on disk

Description

POSIX documents a way of ensuring data is actually sync'ed on permanent storage through fsync(), fdatasync() and aio_fsync().

This way, previously written data, and/or modified meta-data, are guaranteed to be actually protected against a reasonably unexpected situation (system crash, power outage ...)

However, when dealing with file entry handling, such as:
 * file creation (open(O_CREAT))
 * file renaming (rename())
 * symlinking (symlink())
 * hard-linking (link())
 * etc.
there is no documented way to actually give the same guarantee.

Some implementations (such as the Linux glibc) have a somewhat (badly) documented way:
 * open the container directory in read-only (O_RDONLY)
 * apply fsync() or fdatasync() on it

Please refer to the "fsync()'ing a directory file descriptor" thread on the austin-group-l mailing list for insightful comments on this issue.

Several points were discussed, and these (possibly not fully correct) observations were made:
 * directory entries are not attributes of the files they point to, and can not expect to be synchronized [when fsync'ing the file]
 * tracking relationship between directory entries and file descriptors would be cumbersome (a file may be hard-linked in another directory, then have its initial entry being deleted, for example, or renamed to another location)
 * it is not clear whether a directory can be opened at all using open() (readdir() may be the only allowed interface), and what would be the open flags
 * it is not clear what fsync() on a directory file descriptor would do

Desired Action

Clarify that file meta-data have no relationship with directory entry(ies) on the POSIX side.
Clarify how synchronizing a filename operation can be achieved.

Tags No tags attached.

Attached Files

Relationships

Notes

(0001497)
geoffclare
(manager)
2013-03-19 16:49

It is clear that this bug will need an interpretation, but I'm not quite sure how to handle that, given that it raises several issues and on some the standard is clear whereas on one it is not clear. The standard response templates don't seem to cater for this kind of mixed answer.

I believe it is clear from the descriptions of fsync(), open(), dirfd() and various associated definitions in XBD chapter 3 that:

a. Directory entries are data contained in directory files, not attributes of the files they link to, and therefore an fsync() call on a file is not required to have any effect on any directory entries that exist for that file.

b. A file descriptor for a directory can be obtained by using open() with O_RDONLY or O_SEARCH, or by using dirfd() on a directory stream.

c. Since the fsync() description and associated definitions make no mention of specific file types, the requirements apply to all file types, including directories.

The one thing that is not clear is how the definition of Synchronized I/O Data Integrity Completion applies to modifications to directories, since it is worded in terms of "write" operations. I suggest that we reword the first two paragraphs as:

> For read operations, when the operation has been completed or diagnosed if unsuccessful. The operation is complete only when an image of the data has been successfully transferred to the requesting process. If there were any pending write requests or (if the file is a directory) directory modifications affecting the data to be read at the time that the synchronized read operation was requested, these write requests are successfully transferred prior to reading the data.
>
> For write operations and directory modifications, when the operation has been completed or diagnosed if unsuccessful. The operation is complete only when the written data or (if the file is a directory) modified directory entries have been successfully transferred to storage and all file system information required to retrieve them is successfully transferred.

We should also add something to the APPLICATION USAGE section on the fsync() page.

(0001499)
Konrad_Schwarz
(reporter)
2013-03-20 08:03

The paper "Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem" (http://www.usenix.org/publications/library/proceedings/usenix99/full_papers/mckusick/mckusick.pdf) [^] makes it clear that traditional Unix file systems have no need the interface proposed here, as directory operations are atomic and durable by design.

"Traditionally, filesystem consistency has been maintained across system failures either by using synchronous writes to sequence dependent metadata updates or by using write-ahead logging to atomically group them." (First sentence of the Abstract.)

Before burdening writers of portable applications with the interface proposed by this Defect Report, I think it would be worthwhile to find out which file systems break with tradition in this regard.

As the proposed change breaks existing applications -- durability of file system modifications now requires synchronization of directories in addition to synchronization of file system data -- I think it worthwhile to consider an alternative resolution, namely that directory changes are always atomic & durable ("synchronous") on a POSIX file system, without any action required by the programmer.

Finally, in light of the research that has gone into this topic, if the committee decides to introduce an interface for synchronizing directory modifications, a pathconf() constant for determining whether the interface is actually necessary for a given path should be added.

By the way, I suspect http://pubs.opengroup.org/onlinepubs/9699919799/xrat/V4_port.html [^] misspells _POSIX_SYNC_IO as _POSIX_SYNCHRONIZED_IO.

(0001542)
geoffclare
(manager)
2013-04-19 15:27

In the April 18 teleconference it was agreed that the standard should mandate that directory operations are always synchronized on conforming file systems, and should include warnings about non-conforming configurations. The proposed changes are as follows.

Changes to XBD...

At page 94 line 2581-2588 section 3.376 change:

> For read, when the operation has been completed or diagnosed if unsuccessful. The read is complete only when an image of the data has been successfully transferred to the requesting process. If there were any pending write requests affecting the data to be read at the time that the synchronized read operation was requested, these write requests are successfully transferred prior to reading the data.
>
> For write, when the operation has been completed or diagnosed if unsuccessful. The write is complete only when the data specified in the write request is successfully transferred and all file system information required to retrieve the data is successfully transferred.

to:

> For read operations, when the operation has been completed or diagnosed if unsuccessful. The operation is complete only when an image of the data has been successfully transferred to the requesting process. If there were any pending write requests or (if the file is a directory) directory modifications affecting the data to be read at the time that the synchronized read operation was requested, these requests are successfully transferred prior to reading the data.
>
> For write operations and directory modification operations, when the operation has been completed or diagnosed if unsuccessful. The operation is complete only when the written data or (if the file is a directory) modified directory entries have been successfully transferred to storage and all file system information required to retrieve them is successfully transferred.

At page 107 line 2859 add a new XBD 4.2 section (and renumber the current 4.2 and all later 4.x sections):

> 4.2 Directory Operations
>
> All file system operations that read a directory or that modify the contents of a directory (for example creating, unlinking, or renaming a file) shall be completed as defined for synchronized I/O data integrity completion (see section 3.376).
>
> < small>Note: Although conforming file systems are required to perform all directory modifications as synchronized I/O operations, some file systems may support non-conforming configurations (for example via mount options) where directory modifications are not synchronized. Applications that rely on directory modifications being synchronized should only be used with such file systems in their conforming configuration(s).</small>

Changes to XSH...

At page 574 line 19833 section aio_fsync() change the APPLICATION USAGE section from:

> None.

to:

> Refer to fdatasync() and fsync().

At page 815 line 27215 section fdatasync() append to the first paragraph:

> If the file is a directory, an implicit fdatasync() is already performed on every I/O operation (see XBD 4.2) and consequently if fdatasync() is called explicitly it shall take no action and shall return the value 0.

At page 815 line 27232 section fdatasync() change the APPLICATION USAGE section from:

> None.

to:

> Although conforming file systems are required to complete all directory modifications as defined for synchronized I/O data integrity completion, some file systems may support non-conforming configurations (for example via mount options) where directory modifications are not synchronized. When the file system is configured in this way, calls to fdatasync() on directories may cause I/O operations to be synchronized, rather than being a no-op.

At page 954 line 31987 section fsync() add a new paragraph to the APPLICATION USAGE section:

> Since conforming file systems are required to complete all directory modifications as defined for synchronized I/O data integrity completion (see XBD 4.2), calling fsync() on a directory only synchronizes the file attributes such as timestamps. However, some file systems may support non-conforming configurations (for example via mount options) where modifications to directory contents are not synchronized. When the file system is configured in this way, calls to fsync() on directories may cause directory contents to be synchronized in addition to file attributes.

Changes to XRAT...

At page 3444 line 115531 add a new XRAT A.4.2 section (and renumber the current A.4.2 and all later A.4.x sections):

> A.4.2 Directory Operations
>
> Earlier versions of this standard did not make clear that all directory modifications are performed as synchronized I/O operations, although that is the historical behavior and was always intended. Applications have no need to call fdatasync() or fsync() on a directory unless they want to synchronize the file attributes (using fsync()), provided the directory is on a conforming file system. However, since applications may wish to use fdatasync() or fsync() to synchronize directory modifications on non-conforming file systems, implementations are required to support fdatasync() on directories as a no-op on conforming file systems.

(0001545)
eggert (reporter)
2013-04-25 02:22

The proposed changes have caused some consternation on the Glibc developer list (see the thread containing < http://sourceware.org/ml/libc-alpha/2013-04/msg00630.html>), [^] and this suggests that the proposal needs further clarification and/or rewording.

I expect that many modern POSIX implementations fail to conform to the proposed change. Systems with soft updates (as described in the 1999 McKusick & Ganger paper), for example, typically don't guarantee that rename operations are durable: all that's guaranteed is that the file system is consistent after a reboot. Certainly Ext4, XFS, and BTRFS fail to conform to the proposed wording when operating in the default mode, and I expect that many other modern file systems are similar.

If it's really the intent to disallow many (most?) reasonably-high-performance file systems when operating in their default mode, the standard should clearly say so, at least in the rationale somewhere. I'm hoping that this is not the intent, though, as it would create an uncomfortably wide gap between what POSIX requires and what most modern systems actually do.

Instead, I suggest that the POSIX standard and/or rationale be rewritten to more clearly distinguish between atomicity and durability, to say that directory operations must be atomic but not necessarily durable, and to say that fdatasync and fsync can be used to establish the durability of directory operations.

My source for the abovementioned info about Ext4, XFS, and
BTRFS is: Ren K, Gibson G. TableFS: enhancing metadata
efficiency in the local file system. CMU Parallel Data
Laboratory, CMU-PDL-12-110 (September 2012)
< http://www.istc-cc.cmu.edu/publications/papers/2012/CMU-PDL-12-110.pdf>. [^]

(0001548)
geoffclare
(manager)
2013-04-25 09:09
edited on: 2013-04-25 15:10

Suggested Interpretation response

The standard does not speak to the issue of filename synchronization,
and as such no conformance distinction can be made between alternative
implementations based on this. This is being referred to the sponsor.

Rationale:

Although applications can call fsync() to synchronize a directory,
historically this was not necessary (except to synchronize attributes
of the directory such as timestamps) and there are a large number of
existing applications that call fsync() or fdatasync() on files they
create but not on the directories those files are created in. These
applications should not have to change to explicitly synchronize
directories.

In answer to the specific points raised in this request:

a. Directory entries are data contained in directory files, not
attributes of the files they link to, and therefore an fsync()
call on a file is not required to have any effect on any directory
entries that exist for that file.

b. A file descriptor for a directory can be obtained by using
open() with O_RDONLY or O_SEARCH, or by using dirfd() on a
directory stream.

c. Since the fsync() description and associated definitions make no
mention of specific file types, the requirements apply to all file
types, including directories.

d. It is not clear how the definition of Synchronized I/O Data
Integrity Completion applies to modifications to directories, since
it is worded in terms of "write" operations.

Notes to the Editor (not part of this interpretation):

Make the changes described in Note: 0001542.

(0001550)
joerg (reporter)
2013-04-25 11:49

The historical UNIXv6 filesystem did not have rename(), but
UFS from *BSD that started in 1981 did and even the original
implementaion did not grant a rename() to be synchronous but
rather just atomic. In case of a crash, either the old or
the new state was recovered.

The concept at that time already was to order the disk operations
in a way that grants a consistent state but not necessarily a
specific operation to be on the background storage.

Newer concepts like ZFS write copies of meta data that forms a new
tree and a new state appears as the stable state on the background
storage after the related new suberblock (ZFS calls it Überblock)
has been written.

(0001567)
ajosey (manager)
2013-05-03 09:14

[Proposal from Paul Eggert 2 May 2013]

Well, here's a first cut. I'm sure it could use
improvement. In particular, I don't like that last
paragraph....

Changes to XBD

After XBD page 94 line 2613, insert:

> For the purpose of this definition, an operation that reads or
> searches a directory is considered to be a read operation, an
> operation that modifies a directory is considered to be a write

operation, and a directory's entries are considered to be the data read or written.

After page 107 line 2885 add a new XBD 4.2 section (and renumber the current 4.2 and all later 4.x sections):

4.2 Directory Operations

All file system operations that read or search a directory or that modify the contents of a directory (for example creating, unlinking, or renaming a file) shall operate atomically. That is, each operation shall either have its entire effect and succeed, or shall not affect the file system and shall fail. Furthermore, these operations shall be serializable, that is, the state of the file system and of the results of each operation shall always be values that would be obtained if the operations were executed one after the other.

Changes to XSH

At page 579 line 19973 section aio_fsync() change the APPLICATION USAGE section from:

None.

to:

Refer to fdatasync() and fsync().

At page 821 line 27587 section fdatasync() change the APPLICATION USAGE section from:

None.

to:

An application that modifies a directory, e.g., by creating a file in the directory, can invoke fdatasync() on the directory to ensure that the directory's entries are synchronized.

After page 963 line 32522 section fsync() add a new paragraph to the APPLICATION USAGE section:

An application that modifies a directory, e.g., by creating a file in the directory, can invoke fsync() on the directory to ensure that the directory's entries and metadata are synchronized.

Changes to XRAT

After page 3472 line 117096 add a new XRAT A.4.2 section (and renumber the current A.4.2 and all later A.4.x sections):

A.4.2 Directory Operations

Earlier versions of this standard did not make clear that directory modifications are performed atomically and serially, although that is the historical behavior and was always intended. Earlier versions also did not specify the behavior of fdatasync() or fsync() on directories.

Although directory operations are atomic and serializable, they are not necessarily durable. An application that require a directory modification to be durable should use fdatasync() or fsync() on the directory.

It is unspecified whether a directory modification results in a consistent data structure in the storage device associated with the file system, even if fdatasync() or fsync() is used. Some operations, such as rename(), can affect more than one directory, whereas fdatasync() and fsync() can affect at most one directory at a time.

(0001600)
geoffclare
(manager)

Latest proposal, based on Note: 0001567 and email sequence 19026. Page and line numbers are for the 2013 edition.

Changes to XBD...

2013-05-10 11:08

After page 94 line 2613, insert a new paragraph:

For the purpose of this definition, an operation that reads or searches a directory is considered to be a read operation, an operation that modifies a directory is considered to be a write operation, and a directory's entries are considered to be the data read or written.

After page 107 line 2884 add a new XBD 4.2 section (and renumber the current 4.2 and all later 4.x sections):

4.2 Directory Operations

All file system operations that read or search a directory or that modify the contents of a directory (for example creating, unlinking, or renaming a file) shall operate atomically. That is, each operation shall either have its entire effect and succeed, or shall not affect the file system and shall fail. Furthermore, these operations shall be serializable; that is, the state of the file system and of the results of each operation shall always be values that would be obtained if the operations were executed one after the other. If the file system is accessed via a memory cache, these requirements shall apply both to the file system state in the cache and to the file system state on the underlying storage.

If an application creates a regular file, writes to it, and then calls fdatasync(), fsync(), or aio_fsync() on it, the directory entry for the filename used to create the file shall be transferred to storage no later than the file contents.

< small>Note: Although conforming file systems are required to perform all directory modifications as described above, some file systems may support non-conforming configurations (for example via mount options) for which this is not the case. Applications that synchronize regular files but do not explicitly synchronize directories after modifying them should only be used with such file systems in their conforming configuration(s).</small>

Changes to XSH...

At page 579 line 19973 section aio_fsync() change the APPLICATION USAGE section from:

None.

to:

Refer to fdatasync() and fsync().

At page 821 line 27587 section fdatasync() change the APPLICATION USAGE section from:

None.

to:

An application that modifies a directory, e.g., by creating a file in the directory, can invoke fdatasync() on the directory to ensure that the directory's entries are synchronized, although for most applications this should not be necessary (see XBD 4.2).

After page 963 line 32522 section fsync() add a new paragraph to the APPLICATION USAGE section:

An application that modifies a directory, e.g., by creating a file in the directory, can invoke fsync() on the directory to ensure that the directory's entries and file attributes are synchronized. For most applications, synchronizing the directory's entries should not be necessary (see XBD 4.2).

Changes to XRAT...

After page 3472 line 117096 add a new XRAT A.4.2 section (and renumber the current A.4.2 and all later A.4.x sections):

A.4.2 Directory Operations

Earlier versions of this standard did not make clear that directory modifications are performed atomically and serially, although that is the historical behavior and was always intended. Earlier versions also did not specify the behavior of aio_fsync(), fdatasync() or fsync() on directories.

Although directory operations are atomic and serializable, they are not necessarily durable. An application that requires a directory modification to be durable should use fdatasync() or fsync() (or aio_fsync()) on the directory. However, the intention of the requirements for directory modifications is that most applications should not need to do this. For example, a common method of updating a file is to create a new temporary file, call fdatasync() or fsync() to synchronize the new file, and then use rename() to replace the old file with the new file. If a crash occurs after the rename(), then the file being updated will have either its old contents or its new contents on the storage device when the system is rebooted. An application need only synchronize the directory if it wants to be sure the updated file will have its new contents on the storage device.

It is unspecified whether a directory modification results in a consistent data structure in the storage device associated with the file system, even if aio_fsync(), fdatasync() or fsync() is used on directories. Some operations, such as rename(), can affect more than one directory, whereas these synchronization calls can affect at most one directory at a time. If the file system is inconsistent after a crash it is usually automatically checked and repaired when the system is rebooted, or can be repaired manually using a utility such as fsck.

(0001603)
eggert (reporter)
2013-05-14 16:15

In looking at the latest proposed change, I still see problems, and have further suggestions.

1. After discussing atomicity and serializability, the proposed 4.2 now says:

If the file system is accessed via a memory cache, these requirements shall apply both to the file system state in the cache and to the file system state on the underlying storage.

This additional requirement on underlying storage appears to be too strong. If I understand it correctly, it would require that the underlying storage is always the consistent result of serial atomic operations, and would require that sequences of operations be serializable even if the operations apply to different file systems. I doubt whether modern operating systems behave this way.

This sentence was apparently prompted by Geoff Clare's comment in < http://article.gmane.org/gmane.comp.standards.posix.austin.general/7348> [^] that the proposed 4.2 could all be read as applying only to the cache. But that is the intent of proposed 4.2 -- the proposal should not be read as applying to underlying storage. The discussion of underlying storage should be orthogonal to the proposed 4.2.

I suggest rewording this sentence to make this clear, perhaps something like the following:

If the file system is accessed via a memory cache, these requirements shall apply to the file system state in the cache. These requirements are in addition to the requirements for synchronized input and output.

If there is also a need to clarify how synchronized input and output work, perhaps we should add a new paragraph about it.

2. quoting text added to the proposed change:

If an application creates a regular file, writes to it, and then calls fdatasync(), fsync(), or aio_fsync() on it, the directory entry for the filename used to create the file shall be transferred to storage no later than the file contents.

This quoted text has several issues:

2A. I'm not sure that practical POSIXish file systems guarantee this property. Has this been checked?

2B. The quoted text seems to be catering to the following scenario (error-checking omitted):

```
int fd = open ("temp", O_CREAT|O_WRONLY|O_EXCL|..., ...);
write (fd, buf, sizeof buf);
fsync (fd);
close (fd);
rename ("temp", "permanent");
```

But the quoted text doesn't suffice for this scenario, as the application must invoke fsync/fdatasync/etc on the parent directory for the 'rename' to be synchronized and for the change to be committed. And if the application does synchronize the parent directory, then the above-quoted requirement won't help the application -- so why is the requirement helpful?

2C. Even if some sort of requirement along these lines is needed, the quoted text is too strong. Surely the application doesn't need the directory entry to be transferred to storage before the file contents; it could be transferred afterwards. All that'd be needed is that the directory entry be transferred to storage before fsync() returns.

2D. The quoted text doesn't clearly state what happens when other operations intervene. For example, suppose the directory entry is unlinked or renamed after the file is created but before it is written or fsynced. Surely there's no intent that the creating directory entry must be synchronized to storage in these cases.

A simple way to work around these problems is to omit the quoted text.

3. In the <small>Note:

> Applications that synchronize regular files but do not explicitly synchronize directories after modifying them should only be used with such file systems in their conforming configuration(s).

If a file system has a non-conforming configuration, all bets are off: synchronization of any sort is unreliable. Unfortunately the above-quoted text could be misread to imply that although the mentioned synchronization doesn't work some other forms of synchronization do work. I suggest stating the point more generally instead. Perhaps something like this?

> Applications that are used on non-conforming file systems cannot rely on files being synchronized properly.

4. I thought of a new issue.

There's no way, even with this proposal, to synchronize symbolic links. Perhaps we should just clarify this by appending the following after XBD page 94 line 2613:

> The standard provides no way to synchronize the contents or attributes of a symbolic link.

Another option is to fix the symlink problem -- for example, I think GNU/Linux has a way to do this, with its O_PATH flag. But that's a bigger deal.

(0001618)
geoffclare
(manager)
2013-05-21 10:07
edited on: 2013-05-22 10:14

New proposal which is a modified version of Note: 0001600 based on subsequent email discussion.

Page and line numbers are for the 2013 edition.

Changes to XBD...

After page 94 line 2613, insert a new paragraph:

> For the purpose of this definition, an operation that reads or searches a directory is considered to be a read operation, an

operation that modifies a directory is considered to be a write operation, and a directory's entries are considered to be the data read or written.

The standard provides no way to synchronize the contents or attributes of a symbolic link.

After page 107 line 2884 add a new XBD 4.2 section (and renumber the current 4.2 and all later 4.x sections):

4.2 Directory Operations

All file system operations that read or search a directory or that modify the contents of a directory (for example creating, unlinking, or renaming a file) shall operate atomically. That is, each operation shall either have its entire effect and succeed, or shall not affect the file system and shall fail. Furthermore, these operations shall be serializable; that is, the state of the file system and of the results of each operation shall always be values that would be obtained if the operations were executed one after the other.

After page 107 line 2939 add a new XBD 4.8 section (and renumber the remaining 4.x sections):

4.8 File System Cache

If the file system is accessed via a memory cache, file-related requirements stated in the rest of this standard shall apply to the cache, except where explicitly stated otherwise: this includes directory atomicity and serializability requirements (see XBD 4.2), file times update requirements (see XBD 4.10), and read-write serializability requirements (see write()). Cache entries shall be transferred to the underlying storage as the result of successful calls to fdatasync(), fsync(), or aio_fsync(), and may be transferred to storage automatically at other times. Such transfers shall be atomic, with minimum units being directory entries (for directory contents), aligned data blocks of the fundamental file system block size (for regular-file contents; see <sys/statvfs.h>), and all attributes of a single file (for file attributes).

< small>Note: If the system crashes before the cache is fully transferred, later operations' effects may be present in storage with earlier effects missing.</small>

< small>Note: Operations that create or modify multiple directory entries, aligned data blocks, or file attributes (e.g., mkdir(), rename(), write() with large buffer size, open() with O_CREAT) may have only part of their effects transferred to storage, and after a crash these operations may appear to have been only partly done, with the parts not necessarily done in any order. For example, only the second half of a write() may be transferred; or rename("a","b") may result in "b" being created without "a" being removed.</small>

< small>Note: Although conforming file systems are required to perform all caching as described above, some file systems may support non-conforming configurations (for example via mount options) for which this is not the case. Applications that are used on non-conforming file systems cannot rely on files being synchronized properly.</small>

Changes to XSH...

At page 579 line 19973 section aio_fsync() change the APPLICATION USAGE section from:

None.

to:

Refer to fdatasync() and fsync().

At page 821 line 27587 section fdatasync() change the APPLICATION USAGE section from:

None.

to:

An application that modifies a directory, e.g., by creating a file in the directory, can invoke fdatasync() on the directory to ensure that the directory's entries are synchronized, although for most applications this should not be necessary (see XBD 4.8).

After page 963 line 32522 section fsync() add a new paragraph to the APPLICATION USAGE section:

An application that modifies a directory, e.g., by creating a file in the directory, can invoke fsync() on the directory to ensure that the directory's entries and file attributes are synchronized. For most applications, synchronizing the directory's entries should not be necessary (see XBD 4.8).

Changes to XRAT...

After page 3472 line 117096 add new XRAT A.4.2 and A.4.8 sections (and renumber the current A.4.2 and all later A.4.x sections).

A.4.2 Directory Operations

Earlier versions of this standard did not make clear that directory modifications are performed atomically and serially, although that is the historical behavior and was always intended.

A.4.8 File System Cache

Earlier versions of this standard did not specify the behavior of aio_fsync(), fdatasync() or fsync() on directories, nor did they specify constraints on the underlying storage in the absence of calls to aio_fsync, fdatasync() or fsync().

Although directory operations are atomic and serializable, they are not necessarily durable. An application that requires a directory modification to be durable should use fdatasync() or fsync() (or aio_fsync()) on the directory. However, the intention of the requirements for directory modifications is that most applications should not need to do this. For example, a common method of updating a file is to create a new temporary file, call fdatasync() or fsync() to synchronize the new file, and then use rename() to replace the old file with the new file. If a crash occurs after the rename(), then the file being updated will have either its old contents or its new contents on the storage device when the system is rebooted. An application needs to synchronize the directory only if it wants to be sure the updated file will have its new contents on the storage device.

Some operations, such as rename(), can affect more than one directory, whereas synchronization calls such as fsync() can affect at most one directory at a time. Two calls to fsync() may be needed after a rename() to ensure its durability.

If the file system is inconsistent after a crash it is usually automatically checked and repaired when the system is rebooted, or can be repaired manually using a utility such as fsck.

If an unrecoverable I/O error occurs when cache is transferred to storage, this standard provides no way for applications to discover the error reliably. Implementations are encouraged to report such errors on subsequent reads of the storage.

(0001624)
geoffclare
(manager)
2013-05-23 15:26
edited on: 2013-05-23 15:27

In the May 23 teleconference we agreed that we should ask for input from file system developers (specifically, Mark B. to ask the AIX devs, Jim P. to ask the Solaris devs, and Andrew to contact Apple and HP for their input; input from other file system developers would also be welcome).

The questions for the file system developers are:

1. Do you believe your file system(s) conform to the proposed requirements in [Note: 0001618](http://austingroupbugs.net/view.php?id=672)?

2. These requirements currently allow for the possibility that on rename("a/b", "c/d"), the removal of "b" from directory "a"

could be transferred to storage before the creation of "d" in directory "c" is transferred. Thus, if a crash occurs between these two transfers, this would result in neither "a/b" nor "c/d" existing on recovery. Is this correct for your file system(s), or do you handle renames in such a way as to ensure that this can't happen?

(0001691)
jim_pugsley
(manager)
2013-08-01 10:25

1. Solaris devs believe that UFS conforms to the proposed requirements.

2. UFS rename() is atomic, either both directories are modified or neither is.

Issue History

Date Modified	Username	Field	Change
2013-03-19 15:45	xroche	New Issue	
2013-03-19 15:45	xroche	Status	New => Under Review
2013-03-19 15:45	xroche	Assigned To	=> ajosey
2013-03-19 15:45	xroche	Name	=> Xavier Roche
2013-03-19 16:49	geoffclare	Note Added: 0001497	
2013-03-20 01:28	nialldouglas	Note Added: 0001498	
2013-03-20 01:30	nialldouglas	Note Deleted: 0001498	
2013-03-20 08:03	Konrad_Schwarz	Note Added: 0001499	
2013-04-09 06:51	xroche	Issue Monitored: xroche	
2013-04-19 15:27	geoffclare	Note Added: 0001542	
2013-04-25 02:22	eggert	Note Added: 0001545	
2013-04-25 03:40	eggert	Issue Monitored: eggert	
2013-04-25 09:09	geoffclare	Note Added: 0001548	
2013-04-25 09:14	geoffclare	Note Edited: 0001548	
2013-04-25 11:49	joerg	Note Added: 0001550	
2013-04-25 15:10	geoffclare	Note Edited: 0001548	
2013-05-03 09:14	ajosey	Note Added: 0001567	
2013-05-10 11:08	geoffclare	Note Added: 0001600	
2013-05-14 16:15	eggert	Note Added: 0001603	
2013-05-21 10:07	geoffclare	Note Added: 0001618	
2013-05-22 10:13	geoffclare	Note Edited: 0001618	
2013-05-22 10:14	geoffclare	Note Edited: 0001618	
2013-05-23 15:26	geoffclare	Note Added: 0001624	
2013-05-23 15:27	geoffclare	Note Edited: 0001624	
2013-07-16 10:08	sascha_silbe	Issue Monitored: sascha_silbe	
2013-08-01 10:25	jim_pugsley	Note Added: 0001691	

Mantis 1.1.6[^]