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 "$X \leftarrow X + Y$" 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*.

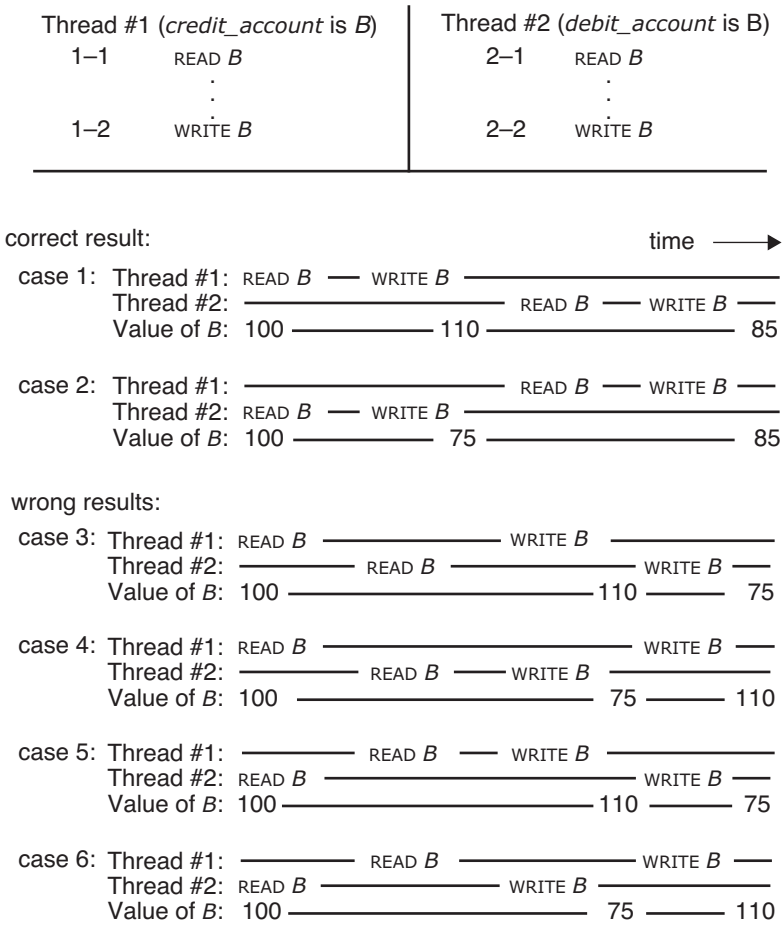

FIGURE 9.2

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:

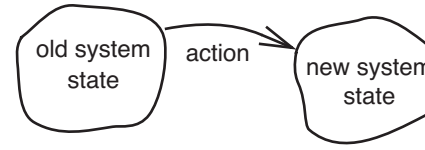

FIGURE 9.3

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.

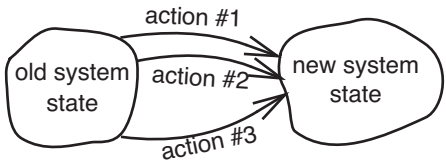

FIGURE 9.4

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

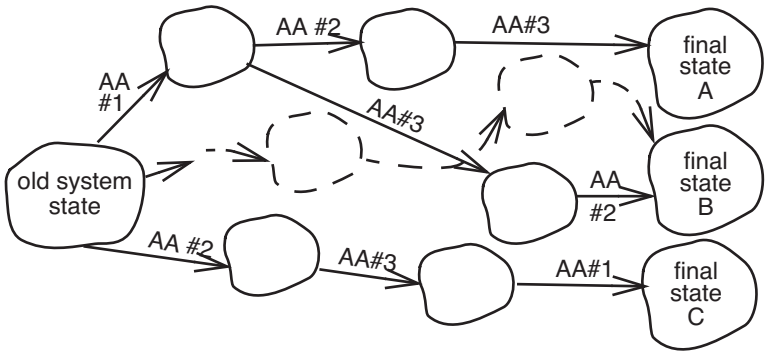

FIGURE 9.5

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