Parallel and distributed applications must take special precautions for handling shared resources. For

example, consider a financial application in which the shared resource is an account record. A thread

running on behalf of a transaction first accesses the account to read the current balance, then updates

the balance, and finally, writes back the new balance. When a thread is interrupted before being able

to complete the three steps of the process, the results of the financial transactions are incorrect if

another thread operating on the same account is allowed to proceed. Another challenge is to deal with

a transaction involving the transfer from one account to another. A system crash after the completion

of the operation on the first account will again lead to an inconsistency – the amount debited from the

first account is not credited to the second.

In these cases, as inmany other similar situations, a multistep operation should be allowed to proceed

to completion without any interruptions, and the operation should be atomic. An important observation

is that such atomic actions should not expose the state of the system until the action is completed. Hiding

the internal state of an atomic action reduces the number of states a system can be in; thus, it simplifies

the design and maintenance of the system. An atomic action is composed of several steps, each of which

may fail; therefore, we have to take additional precautions to avoid exposing the internal state of the

system in case of such a failure.

The discussion of the transaction system suggests that an analysis of atomicity should pay special

attention to the basic operation of updating the value of an object in storage. Even to modify the contents

of amemory location, severalmachine instructions must be executed: load the current value in a register,

modify the contents of the register, and store back the result.

Atomicity cannot be implemented without some hardware support; indeed, the instruction sets of

most processors support the test-and-set instruction, which writes to a memory location and returns the old content of that memory cell as noninterruptible operations. Other architectures support

compare-and-swap, an atomic instruction that compares the contents of a memory location to a given

value and, only if the two values are the same, modifies the contents of that memory location to a given

new value.

Two flavors of atomicity can be distinguished: all-or-nothing and before-or-after atomicity. Allor-

nothing means that either the entire atomic action is carried out, or the system is left in the same

state it was before the atomic action was attempted. In our examples a transaction is either carried out

successfully, or the record targeted by the transaction is returned to its original state. The states of an

all-or-nothing action are shown in Figure 2.12.

To guarantee the all-or-nothing property of an action we have to distinguish preparatory actions

that can be undone from irreversible ones, such as the alteration of the only copy of an object. Such

preparatory actions are as follows: allocation of a resource, fetching a page from secondary storage,

allocation of memory on the stack, and so on. One of the golden rules of data management is never to

change the only copy; maintaining the history of changes and a log of all activities allows us to deal

with system failures and to ensure consistency.

An all-or-nothing action consists of a pre-commit and a post-commit phase; during the former it

should be possible to back up from it without leaving any trace, whereas the latter phase should be

able to run to completion. The transition from the first to the second phase is called a commit point.

During the pre-commit phase all steps necessary to prepare the post-commit phase – for example, check

permissions, swap in main memory all pages that may be needed, mount removable media, and allocate

stack space – must be carried out; during this phase no results should be exposed and no irreversible

actions should be carried out. Shared resources allocated during the pre-commit phase cannot be released

until after the commit point. The commit step should be the last step of an all-or-nothing action.

A discussion of storage models illustrates the effort required to support all-or-nothing atomicity (see

Figure 2.13). The common storage model implemented by hardware is the so-called cell storage, a

collection of cells each capable of holding an object (e.g., the primary memory of a computer where

each cell is addressable). Cell storage does not support all-or-nothing actions. Once the content of a cell

is changed by an action, there is no way to abort the action and restore the original content of the cell.

To be able to restore a previous value we have to maintain a version history for each variable in the

cell storage. The storage model that supports all-or-nothing actions is called journal storage. Now the

cell storage is no longer accessible to the action because the access is mitigated by a storage manager.

In addition to the basic primitives to read an existing value and to write a new value in cell storage,

the storage manager uniquely identifies an action that changes the value in cell storage and, when the action is aborted, is able to retrieve the version of the variable before the action and restore it. When

the action is committed, then the new value should be written to the cell.

Figure 2.13 shows that for a journal storage, in addition to the version histories of all variables

affected by the action, we have to implement a catalog of variables and maintain a record to identify

each new action. Anew action first invokes the Action primitive; at that time an outcome record uniquely

identifying the action is created. Then, every time the action accesses a variable, the version history is

modified. Finally, the action invokes either a Commit or an Abort primitive. In the journal storage model

the action is atomic and follows the state transition diagram in Figure 2.12.

Before-or-after atomicity means that, from the point of view of an external observer, the effect of

multiple actions is as though these actions have occurred one after another, in some order. A stronger

condition is to impose a sequential order among transitions. In our example the transaction acting on

two accounts should either debit the first account and then credit the second one or leave both accounts

unchanged. The order is important because the first account cannot be left with a negative balance.

Atomicity is a critical concept in our efforts to build reliable systems from unreliable components and,

at the same time, to support as much parallelism as possible for better performance. Atomicity allows

us to deal with unforseen events and to support coordination of concurrent activities. The unforseen

event could be a system crash, a request to share a control structure, the need to suspend an activity, and so on; in all these cases we have to save the state of the process or of the entire system to be able

to restart it at a later time.

Because atomicity is required in many contexts, it is desirable to have a systematic approach rather

than an ad hoc one. A systematic approach to atomicity must address several delicate questions:

• How to guarantee that only one atomic action has access to a shared resource at any given time.

• How to return to the original state of the system when an atomic action fails to complete.

• How to ensure that the order of several atomic actions leads to consistent results.

Answers to these questions increase the complexity of the system and often generate additional

problems. For example, access to shared resources can be protected by locks, but when there are

multiple shared resources protected by locks, concurrent activities may deadlock. A lock is a construct

that enforces sequential access to a shared resource; such actions are packaged in the critical sections of

the code. If the lock is not set, a thread first locks the access, then enters the critical section, and finally

unlocks it; a thread that wants to enter the critical section finds the lock set and waits for the lock to be

reset. A lock can be implemented using the hardware instructions supporting atomicity.

Semaphores and monitors are more elaborate structures ensuring serial access. Semaphores force

processes to queue when the lock is set and are released from this queue and allowed to enter the critical

section one by one. Monitors provide special procedures to access the shared data (see Figure 2.14).

The mechanisms for the process coordination we described require the cooperation of all activities, the

same way traffic lights prevent accidents only as long as drivers follow the rules.