- *Deterministic vs. Stochastic Simulation Models.* If a simulation model does not contain any probabilistic (i.e., random) components, it is called *deterministic*; a complicated (and analytically intractable) system of differential equations describing a chemical reaction might be such a model. In deterministic models, the output is "determined" once the set of input quantities and relationships in the model have been specified, even though it might take a lot of computer time to evaluate what it is. Many systems, however, must be modeled as having at least some random input components, and these give rise to *stochastic* simulation models. (For an example of the danger of ignoring randomness in modeling a system, see Sec. 4.7.) Most queueing and inventory systems are modeled stochastically. Stochastic simulation models produce output that is itself random, and must therefore be treated as only an estimate of the true characteristics of the model; this is one of the main disadvantages of simulation (see Sec. 1.8) and is dealt with in Chaps. 9 through 12 of this book.
- *Continuous vs. Discrete Simulation Models.* Loosely speaking, we define *discrete* and *continuous* simulation models analogously to the way discrete and continuous systems were defined above. More precise definitions of discrete (event) simulation and continuous simulation are given in Secs. 1.3 and 13.3, respectively. It should be mentioned that a discrete model is not always used to model a discrete system, and vice versa. The decision whether to use a discrete or a continuous model for a particular system depends on the specific objectives of the study. For example, a model of traffic flow on a freeway would be discrete if the characteristics and movement of individual cars are important. Alternatively, if the cars can be treated "in the aggregate," the flow of traffic can be described by differential equations in a continuous model. More discussion on this issue can be found in Sec. 5.2, and in particular in Example 5.2.

The simulation models we consider in the remainder of this book, except for those in Chap. 13, will be discrete, dynamic, and stochastic and will henceforth be called *discrete-event simulation models*. (Since deterministic models are a special case of stochastic models, the restriction to stochastic models involves no loss of generality.)

1.3 DISCRETE-EVENT SIMULATION

Discrete-event simulation concerns the modeling of a system as it evolves over time by a representation in which the state variables change instantaneously at separate points in time. (In more mathematical terms, we might say that the system can change at only a *countable* number of points in time.) These points in time are the ones at which an event occurs, where an *event* is defined as an instantaneous occurrence that may change the state of the system. Although discrete-event simulation could conceptually be done by hand calculations, the amount of data that must be stored and manipulated for most real-world systems dictates that discrete-event simulations be done on a digital computer. (In Sec. 1.4.2 we carry out a small hand simulation, merely to illustrate the logic involved.)

EXAMPLE 1.1. Consider a service facility with a single server—e.g., a one-operator barbershop or an information desk at an airport—for which we would like to estimate the (expected) average delay in queue (line) of arriving customers, where the delay in queue of a customer is the length of the time interval from the instant of his arrival at the facility to the instant he begins being served. For the objective of estimating the average delay of a customer, the state variables for a discrete-event simulation model of the facility would be the status of the server, i.e., either idle or busy, the number of customers waiting in queue to be served (if any), and the time of arrival of each person waiting in queue. The status of the server is needed to determine, upon a customer's arrival, whether the customer can be served immediately or must join the end of the queue. When the server completes serving a customer, the number of customers in the queue is used to determine whether the server will become idle or begin serving the first customer in the queue. The time of arrival of a customer is needed to compute his delay in queue, which is the time he begins being served (which will be known) minus his time of arrival. There are two types of events for this system: the arrival of a customer and the completion of service for a customer, which results in the customer's departure. An arrival is an event since it causes the (state variable) server status to change from idle to busy or the (state variable) number of customers in the queue to increase by 1. Correspondingly, a departure is an event because it causes the server status to change from busy to idle or the number of customers in the queue to decrease by 1. We show in detail how to build a discrete-event simulation model of this single-server queueing system in Sec. 1.4.

In the above example both types of events actually changed the state of the system, but in some discrete-event simulation models events are used for purposes that do not actually effect such a change. For example, an event might be used to schedule the end of a simulation run at a particular time (see Sec. 1.4.6) or to schedule a decision about a system's operation at a particular time (see Sec. 1.5) and might not actually result in a change in the state of the system. This is why we originally said that an event *may* change the state of a system.

1.3.1 Time-Advance Mechanisms

Because of the dynamic nature of discrete-event simulation models, we must keep track of the current value of simulated time as the simulation proceeds, and we also need a mechanism to advance simulated time from one value to another. We call the variable in a simulation model that gives the current value of simulated time the *simulation clock*. The unit of time for the simulation clock is never stated explicitly when a model is written in a general-purpose language such as C, and it is assumed to be in the same units as the input parameters. Also, there is generally no relationship between simulated time and the time needed to run a simulation on the computer.

Historically, two principal approaches have been suggested for advancing the simulation clock: *next-event time advance* and *fixed-increment time advance*. Since the first approach is used by all major simulation software and by most people programming their model in a general-purpose language, and since the second is a special case of the first, we shall use the next-event time-advance approach for all discrete-event simulation models discussed in this book. A brief discussion of fixed-increment time advance is given in App. 1A (at the end of this chapter).

With the next-event time-advance approach, the simulation clock is initialized to zero and the times of occurrence of future events are determined. The simulation clock is then advanced to the time of occurrence of the *most imminent* (first) of these future events, at which point the state of the system is updated to account for the fact that an event has occurred, and our knowledge of the times of occurrence of future events is also updated. Then the simulation clock is advanced to the time of the (new) most imminent event, the state of the system is updated, and future event times are determined, etc. This process of advancing the simulation clock from one event time to another is continued until eventually some prespecified stopping condition is satisfied. Since all state changes occur only at event times for a discrete-event simulation model, periods of inactivity are skipped over by jumping the clock from event time to event time. (Fixed-increment time advance does not skip over these inactive periods, which can eat up a lot of computer time; see App. 1A.) It should be noted that the successive jumps of the simulation clock are generally variable (or unequal) in size.

EXAMPLE 1.2. We now illustrate in detail the next-event time-advance approach for the single-server queueing system of Example 1.1. We need the following notation:

t_i = time of arrival of the ith customer ($t_0 = 0$)
$A_i = t_i - t_{i-1}$ = interarrival time between $(i-1)$st and ith arrivals of customers
S_i = time that server actually spends serving ith customer (exclusive of customer's delay in queue)
D_i = delay in queue of ith customer
$c_i = t_i + D_i + S_i$ = time that ith customer completes service and departs
e_i = time of occurrence of ith event of any type (ith value the simulation clock takes on, excluding the value $e_0 = 0$)

Each of these defined quantities will generally be a random variable. Assume that the probability distributions of the interarrival times $A_1, A_2, \ldots$ and the service times $S_1, S_2, \ldots$ are known and have cumulative distribution functions (see Sec. 4.2) denoted by F_A and F_S, respectively. (In general, F_A and F_S would be determined by collecting data from the system of interest and then specifying distributions consistent with these data using the techniques of Chap. 6.) At time $e_0 = 0$ the status of the server is idle, and the time t_1 of the first arrival is determined by generating A_1 from F_A (techniques for generating random observations from a specified distribution are discussed in Chap. 8) and adding it to 0. The simulation clock is then advanced from e_0 to the time of the next (first) event, $e_1 = t_1$. (See Fig. 1.2, where the curved arrows represent advancing the simulation clock.) Since the customer arriving at time t_1 finds the server idle, she immediately enters service and has a delay in queue of $D_1 = 0$ and the status of the server is changed from idle to busy. The time, c_1, when the arriving customer will complete service is computed by generating S_1 from F_S and adding it to t_1. Finally, the time of the second arrival, t_2, is computed as $t_2 = t_1 + A_2$, where A_2 is generated from F_A. If $t_2 < c_1$, as depicted in Fig. 1.2, the simulation clock is advanced from e_1 to the time of the next event, $e_2 = t_2$. (If c_1 were less than t_2, the clock would be advanced from e_1 to c_1.) Since the customer arriving at time t_2 finds the server already busy, the number of customers in the queue is increased from 0 to 1 and the time of arrival of this customer is recorded; however, his service time S_2 is not generated at this time. Also, the time of the third arrival, t_3, is computed as $t_3 = t_2 + A_3$. If $c_1 < t_3$, as depicted in the figure, the simulation clock is advanced from e_2 to the time of the next event, $e_3 = c_1$, where the customer

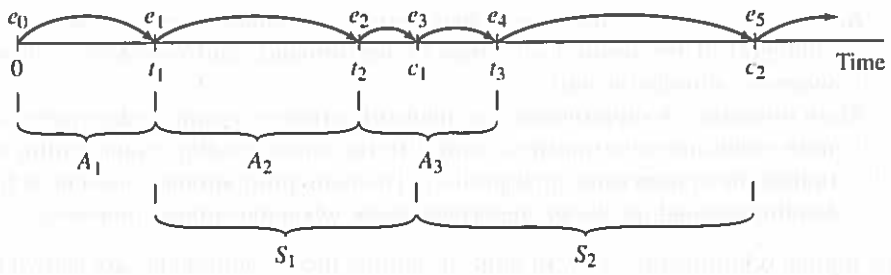


FIGURE 1.2
The next-event time-advance approach illustrated for the single-server queueing system.

completing service departs, the customer in the queue (i.e., the one who arrived at time t_2) begins service and his delay in queue and service-completion time are computed as $D_2 = c_1 - t_2$ and $c_2 = c_1 + S_2$ (S_2 is now generated from F_S), and the number of customers in the queue is decreased from 1 to 0. If $t_3 < c_2$, the simulation clock is advanced from e_3 to the time of the next event, $e_4 = t_3$, etc. The simulation might eventually be terminated when, say, the number of customers whose delays have been observed reaches some specified value.

1.3.2 Components and Organization of a Discrete-Event Simulation Model

Although simulation has been applied to a great diversity of real-world systems, discrete-event simulation models all share a number of common components and there is a logical organization for these components that promotes the programming, debugging, and future changing of a simulation model's computer program. In particular, the following components will be found in most discrete-event simulation models using the next-event time-advance approach programmed in a general-purpose language:

System state: The collection of state variables necessary to describe the system at a particular time
Simulation clock: A variable giving the current value of simulated time
Event list: A list containing the next time when each type of event will occur
Statistical counters: Variables used for storing statistical information about system performance
Initialization routine: A subprogram to initialize the simulation model at time 0
Timing routine: A subprogram that determines the next event from the event list and then advances the simulation clock to the time when that event is to occur
Event routine: A subprogram that updates the system state when a particular type of event occurs (there is one event routine for each event type)
Library routines: A set of subprograms used to generate random observations from probability distributions that were determined as part of the simulation model

Report generator: A subprogram that computes estimates (from the statistical counters) of the desired measures of performance and produces a report when the simulation ends

Main program: A subprogram that invokes the timing routine to determine the next event and then transfers control to the corresponding event routine to update the system state appropriately. The main program may also check for termination and invoke the report generator when the simulation is over.

The logical relationships (flow of control) among these components are shown in Fig. 1.3. The simulation begins at time 0 with the main program invoking the initialization routine, where the simulation clock is set to zero, the system state and the statistical counters are initialized, and the event list is initialized. After control has been returned to the main program, it invokes the timing routine to determine which type of event is most imminent. If an event of type i is the next to occur, the simulation clock is advanced to the time that event type i will occur

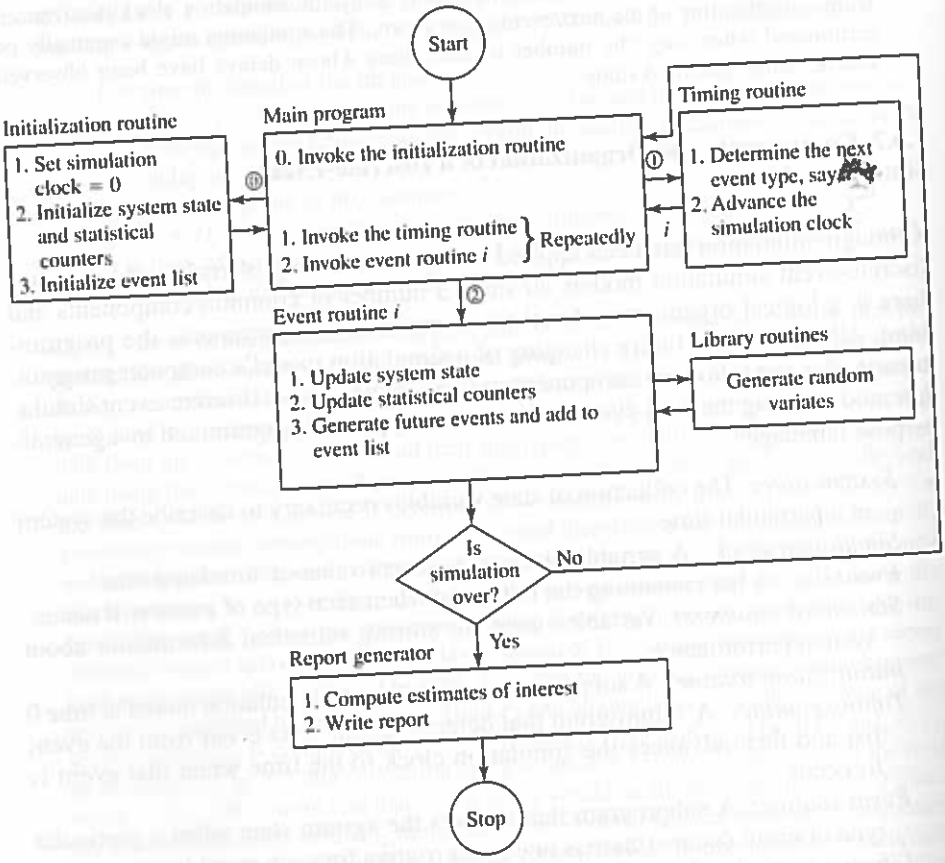


FIGURE 1.3
Flow of control for the next-event time-advance approach.

and control is returned to the main program. Then the main program invokes event routine i, where typically three types of activities occur: (1) The system state is updated to account for the fact that an event of type i has occurred; (2) information about system performance is gathered by updating the statistical counters; and (3) the times of occurrence of future events are generated, and this information is added to the event list. Often it is necessary to generate random observations from probability distributions in order to determine these future event times; we will refer to such a generated observation as a *random variate*. After all processing has been completed, either in event routine i or in the main program, a check is typically made to determine (relative to some stopping condition) if the simulation should now be terminated. If it is time to terminate the simulation, the report generator is invoked from the main program to compute estimates (from the statistical counters) of the desired measures of performance and to produce a report. If it is not time for termination, control is passed back to the main program and the main program–timing routine–main program–event routine–termination check cycle is repeated until the stopping condition is eventually satisfied.

Before concluding this section, a few additional words about the system state may be in order. As mentioned in Sec. 1.2, a system is a well-defined collection of *entities*. Entities are characterized by data values called *attributes*, and these attributes are part of the system state for a discrete-event simulation model. Furthermore, entities with some common property are often grouped together in *lists* (or *files* or *sets*). For each entity there is a *record* in the list consisting of the entity's attributes, and the order in which the records are placed in the list depends on some specified rule. (See Chap. 2 for a discussion of efficient approaches for storing lists of records.) For the single-server queueing facility of Examples 1.1 and 1.2, the entities are the server and the customers in the facility. The server has the attribute "server status" (busy or idle), and the customers waiting in queue have the attribute "time of arrival." (The number of customers in the queue might also be considered an attribute of the server.) Furthermore, as we shall see in Sec. 1.4, these customers in queue will be grouped together in a list.

The organization and action of a discrete-event simulation program using the next-event time-advance mechanism as depicted above are fairly typical when programming such simulations in a general-purpose programming language such as C; it is called the *event-scheduling approach* to simulation modeling, since the times of future events are explicitly coded into the model and are scheduled to occur in the simulated future. It should be mentioned here that there is an alternative approach to simulation modeling, called the *process approach*, that instead views the simulation in terms of the individual entities involved, and the code written describes the "experience" of a "typical" entity as it "flows" through the system; programming simulations modeled from the process point of view usually requires the use of special-purpose simulation software, as discussed in Chap. 3. Even when taking the process approach, however, the simulation is actually executed behind the scenes in the event-scheduling logic as described above.

1.4 SIMULATION OF A SINGLE-SERVER QUEUEING SYSTEM

This section shows in detail how to simulate a single-server queueing system such as a one-operator barbershop. Although this system seems very simple compared with those usually of real interest, how it is simulated is actually quite representative of the operation of simulations of great complexity.

In Sec. 1.4.1 we describe the system of interest and state our objectives more precisely. We explain intuitively how to simulate this system in Sec. 1.4.2 by showing a "snapshot" of the simulated system just after each event occurs. Section 1.4.3 describes the language-independent organization and logic of the C code given in Sec. 1.4.4. The simulation's results are discussed in Sec. 1.4.5, and Sec. 1.4.6 alters the stopping rule to another common way to end simulations. Finally, Sec. 1.4.7 briefly describes a technique for identifying and simplifying the event and variable structure of a simulation.

1.4.1 Problem Statement

Consider a single-server queueing system (see Fig. 1.4) for which the interarrival times $A_1, A_2, \ldots$ are *independent and identically distributed* (IID) random variables.

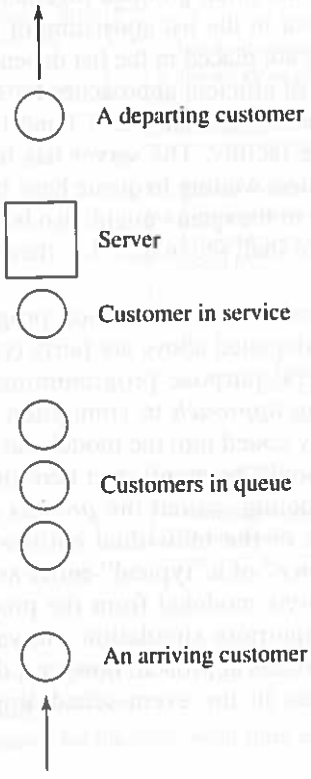


FIGURE 1.4
A single-server queueing system.

("Identically distributed" means that the interarrival times have the same probability distribution.) A customer who arrives and finds the server idle enters service immediately, and the service times $S_1, S_2, \ldots$ of the successive customers are IID random variables that are independent of the interarrival times. A customer who arrives and finds the server busy joins the end of a single queue. Upon completing service for a customer, the server chooses a customer from the queue (if any) in a first-in, first-out (FIFO) manner. (For a discussion of other queue disciplines and queueing systems in general, see App. 1B.)

The simulation will begin in the "empty-and-idle" state; i.e., no customers are present and the server is idle. At time 0, we will begin waiting for the arrival of the first customer, which will occur after the first interarrival time, A_1, rather than at time 0 (which would be a possibly valid, but different, modeling assumption). We wish to simulate this system until a fixed number (n) of customers have completed their delays in queue; i.e., the simulation will stop when the nth customer enters service. Note that the *time* the simulation ends is thus a random variable, depending on the observed values for the interarrival and service-time random variables.

To measure the performance of this system, we will look at estimates of three quantities. First, we will estimate the expected average delay in queue of the n customers completing their delays during the simulation; we denote this quantity by $d(n)$. The word "expected" in the definition of $d(n)$ means this: On a given run of the simulation (or, for that matter, on a given run of the actual system the simulation model represents), the actual average delay observed of the n customers depends on the interarrival and service-time random variable observations that happen to have been obtained. On another run of the simulation (or on a different day for the real system) there would probably be arrivals at different times, and the service times required would also be different; this would give rise to a different value for the average of the n delays. Thus, the average delay on a given run of the simulation is properly regarded as a random variable itself. What we want to estimate, $d(n)$, is the *expected value* of this random variable. One interpretation of this is that $d(n)$ is the average of a large (actually, infinite) number of n-customer average delays. From a single run of the simulation resulting in customer delays $D_1, D_2, \ldots, D_n$, an obvious estimator of $d(n)$ is

$$\hat{d}(n) = \frac{\sum_{i=1}^{n} D_i}{n}$$

which is just the average of the n D_i's that were observed in the simulation [so that $\hat{d}(n)$ could also be denoted by $\overline{D}(n)$]. [Throughout this book, a hat (^) above a symbol denotes an estimator.] It is important to note that by "delay" we do not exclude the possibility that a customer could have a delay of zero in the case of an arrival finding the system empty and idle (with this model, we know for sure that $D_1 = 0$); delays with a value of 0 *are* counted in the average, since if many delays were zero this would represent a system providing very good service, and our output measure should reflect this. One reason for taking the average of the D_i's, as opposed to just looking at them individually, is that they will not have the same distribution (e.g., $D_1 = 0$, but D_2 could be positive), and the average gives us a single composite measure of all

the customers' delays; in this sense, this is not the usual "average" taken in basic statistics, as the individual terms are not independent random observations from the same distribution. Note also that by itself, $\hat{d}(n)$ is an estimator based on a sample of size *1*, since we are making only one complete simulation run. From elementary statistics, we know that a sample of size 1 is not worth much; we return to this issue in Chaps. 9 through 12.

While an estimate of $d(n)$ gives information about system performance from the customers' point of view, the management of such a system may want different information; indeed, since most real simulations are quite complex and may be time-consuming to run, we usually collect many output measures of performance, describing different aspects of system behavior. One such measure for our simple model here is the expected average number of customers in the queue (but not being served), denoted by $q(n)$, where the n is necessary in the notation to indicate that this average is taken over the time period needed to observe the n delays defining our stopping rule. This is a different kind of "average" than the average delay in queue, because it is taken over (continuous) time, rather than over customers (being discrete). Thus, we need to define what is meant by this *time*-average number of customers in queue. To do this, let $Q(t)$ denote the number of customers in queue at time t, for any real number $t \geq 0$, and let $T(n)$ be the time required to observe our n delays in queue. Then for any time t between 0 and $T(n)$, $Q(t)$ is a nonnegative integer. Further, if we let p_i be the expected *proportion* (which will be between 0 and 1) of the time that $Q(t)$ is equal to i, then a reasonable definition of $q(n)$ would be

$$q(n) = \sum_{i=0}^{\infty} ip_i$$

Thus, $q(n)$ is a weighted average of the possible values i for the queue length $Q(t)$, with the weights being the expected proportion of time the queue spends at each of its possible lengths. To estimate $q(n)$ from a simulation, we simply replace the p_i's with estimates of them, and get

$$\hat{q}(n) = \sum_{i=0}^{\infty} i\hat{p}_i \tag{1.1}$$

where $\hat{p}_i$ is the *observed* (rather than expected) proportion of the time *during the simulation* that there were i customers in the queue. Computationally, however, it is easier to rewrite $\hat{q}(n)$ using some geometric considerations. If we let T_i be the *total* time during the simulation that the queue is of length i, then $T(n) = T_0 + T_1 + T_2 + \cdots$ and $\hat{p}_i = T_i/T(n)$, so that we can rewrite Eq. (1.1) above as

$$\hat{q}(n) = \frac{\sum_{i=0}^{\infty} iT_i}{T(n)} \tag{1.2}$$

Figure 1.5 illustrates a possible time path, or *realization*, of $Q(t)$ for this system in the case of $n = 6$; ignore the shading for now. Arrivals occur at times 0.4, 1.6, 2.1, 3.8, 4.0, 5.6, 5.8, and 7.2. Departures (service completions) occur at times 2.4, 3.1, 3.3, 4.9, and 8.6, and the simulation ends at time $T(6) = 8.6$. Remember in looking

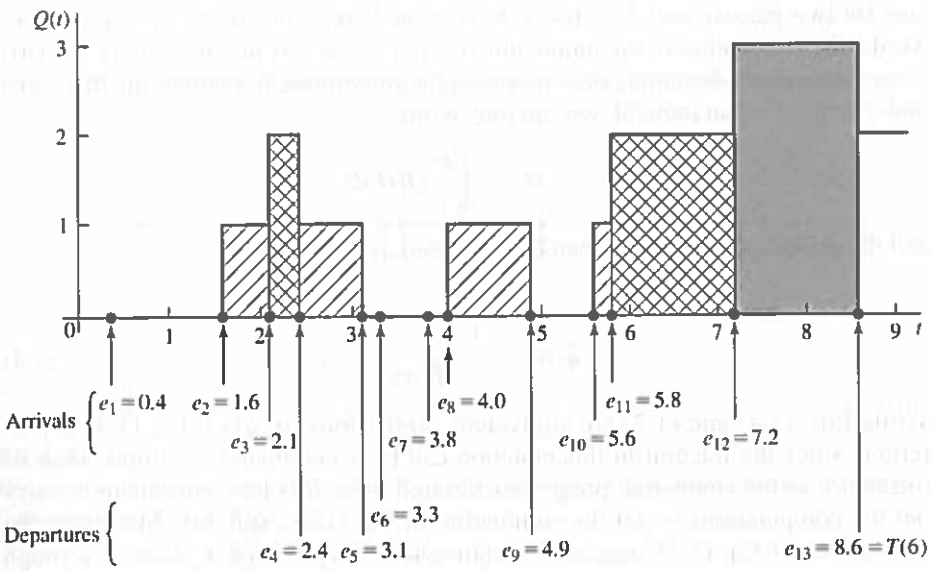


FIGURE 1.5
$Q(t)$, arrival times, and departure times for a realization of a single-server queueing system.

at Fig. 1.5 that $Q(t)$ does not count the customer in service (if any), so between times 0.4 and 1.6 there is one customer in the system being served, even though the queue is empty [$Q(t) = 0$]; the same is true between times 3.1 and 3.3, between times 3.8 and 4.0, and between times 4.9 and 5.6. Between times 3.3 and 3.8, however, the system is empty of customers and the server is idle, as is obviously the case between times 0 and 0.4. To compute $\hat{q}(n)$, we must first compute the T_i's, which can be read off Fig. 1.5 as the (sometimes separated) intervals over which $Q(t)$ is equal to 0, 1, 2, and so on:

$$T_0 = (1.6 - 0.0) + (4.0 - 3.1) + (5.6 - 4.9) = 3.2$$
$$T_1 = (2.1 - 1.6) + (3.1 - 2.4) + (4.9 - 4.0) + (5.8 - 5.6) = 2.3$$
$$T_2 = (2.4 - 2.1) + (7.2 - 5.8) = 1.7$$
$$T_3 = (8.6 - 7.2) = 1.4$$

($T_i = 0$ for $i \geq 4$, since the queue never grew to those lengths in this realization.) The numerator in Eq. (1.2) is thus

$$\sum_{i=0}^{\infty} iT_i = (0 \times 3.2) + (1 \times 2.3) + (2 \times 1.7) + (3 \times 1.4) = 9.9 \qquad (1.3)$$

and so our estimate of the time-average number in queue from this particular simulation run is $\hat{q}(6) = 9.9/8.6 = 1.15$. Now, note that each of the nonzero terms on the right-hand side of Eq. (1.3) corresponds to one of the shaded areas in Fig. 1.5: 1×2.3 is the diagonally shaded area (in four pieces), 2×1.7 is the cross-hatched

area (in two pieces), and 3×1.4 is the screened area (in a single piece). In other words, the summation in the numerator of Eq. (1.2) is just the *area under the $Q(t)$ curve between the beginning and the end of the simulation.* Remembering that "area under a curve" is an integral, we can thus write

$$\sum_{i=0}^{\infty} iT_i = \int_0^{T(n)} Q(t)\, dt$$

and the estimator of $q(n)$ can then be expressed as

$$\hat{q}(n) = \frac{\int_0^{T(n)} Q(t)\, dt}{T(n)} \tag{1.4}$$

While Eqs. (1.4) and (1.2) are equivalent expressions for $\hat{q}(n)$, Eq. (1.4) is preferable since the integral in this equation can be accumulated as simple areas of rectangles as the simulation progresses through time. It is less convenient to carry out the computations to get the summation in Eq. (1.2) explicitly. Moreover, the appearance of Eq. (1.4) suggests a continuous average of $Q(t)$, since in a rough sense, an integral can be regarded as a continuous summation.

The third and final output measure of performance for this system is a measure of how busy the server is. The expected *utilization* of the server is the expected proportion of time during the simulation [from time 0 to time $T(n)$] that the server is busy (i.e., not idle), and is thus a number between 0 and 1; denote it by $u(n)$. From a single simulation, then, our estimate of $u(n)$ is $\hat{u}(n)$ = the *observed* proportion of time during the simulation that the server is busy. Now $\hat{u}(n)$ could be computed directly from the simulation by noting the times at which the server changes status (idle to busy or vice versa) and then doing the appropriate subtractions and division. However, it is easier to look at this quantity as a continuous-time average, similar to the average queue length, by defining the "busy function"

$$B(t) = \begin{cases} 1 & \text{if the server is busy at time } t \\ 0 & \text{if the server is idle at time } t \end{cases}$$

and so $\hat{u}(n)$ could be expressed as the proportion of time that $B(t)$ is equal to 1. Figure 1.6 plots $B(t)$ for the same simulation realization as used in Fig. 1.5 for $Q(t)$. In this case, we get

$$\hat{u}(n) = \frac{(3.3 - 0.4) + (8.6 - 3.8)}{8.6} = \frac{7.7}{8.6} = 0.90 \tag{1.5}$$

indicating that the server was busy about 90 percent of the time during this simulation. Again, however, the numerator in Eq. (1.5) can be viewed as the area under the $B(t)$ function over the course of the simulation, since the height of $B(t)$ is always either 0 or 1. Thus,

$$\hat{u}(n) = \frac{\int_0^{T(n)} B(t)\, dt}{T(n)} \tag{1.6}$$

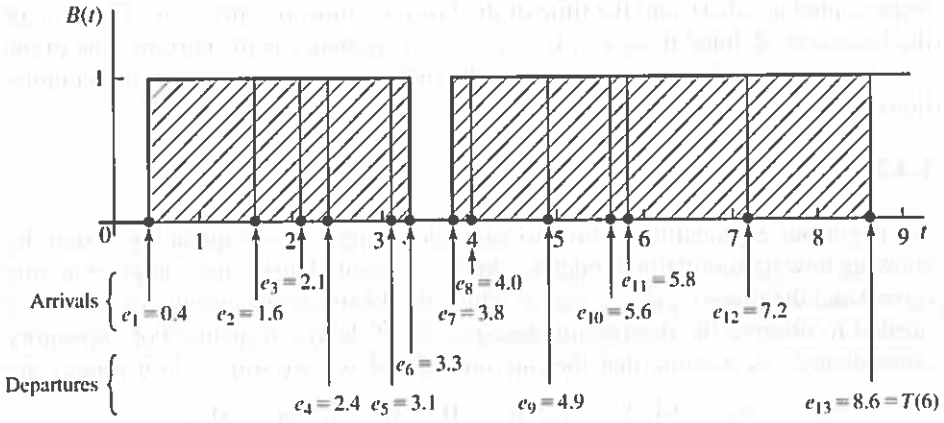


FIGURE 1.6
$B(t)$, arrival times, and departure times for a realization of a single-server queueing system (same realization as in Fig. 1.5).

and we see again that $\hat{u}(n)$ is the continuous average of the $B(t)$ function, corresponding to our notion of utilization. As was the case for $\hat{q}(n)$, the reason for writing $\hat{u}(n)$ in the integral form of Eq. (1.6) is that computationally, as the simulation progresses, the integral of $B(t)$ can easily be accumulated by adding up areas of rectangles. For many simulations involving "servers" of some sort, utilization statistics are quite informative in identifying bottlenecks (utilizations near 100 percent, coupled with heavy congestion measures for the queue leading in) or excess capacity (low utilizations); this is particularly true if the "servers" are expensive items such as robots in a manufacturing system or large mainframe computers in a data-processing operation.

To recap, the three measures of performance are the average delay in queue $\hat{d}(n)$, the time-average number of customers in queue $\hat{q}(n)$, and the proportion of time the server is busy $\hat{u}(n)$. The average delay in queue is an example of a *discrete-time statistic*, since it is defined relative to the collection of random variables $\{D_i\}$ that have a discrete "time" index, $i = 1, 2, \ldots$. The time-average number in queue and the proportion of time the server is busy are examples of *continuous-time statistics*, since they are defined on the collection of random variables $\{Q(t)\}$ and $\{B(t)\}$, respectively, each of which is indexed on the continuous time parameter $t \in [0, \infty)$. (The symbol $\in$ means "contained in." Thus, in this case, t can be any nonnegative real number.) Both discrete-time and continuous-time statistics are common in simulation, and they furthermore can be other than averages. For example, we might be interested in the *maximum* of all the delays in queue observed (a discrete-time statistic), or the *proportion* of time during the simulation that the queue contained at least five customers (a continuous-time statistic).

The events for this system are the arrival of a customer and the departure of a customer (after a service completion); the state variables necessary to estimate $d(n)$, $q(n)$, and $u(n)$ are the status of the server (0 for idle and 1 for busy), the number of customers in the queue, the time of arrival of each customer currently in the queue

(represented as a list), and the time of the last (i.e., most recent) event. The time of the last event, defined to be e_{i-1} if $e_{i-1} \le t < e_i$ (where t is the current time in the simulation), is needed to compute the width of the rectangles for the area accumulations in the estimates of $q(n)$ and $u(n)$.

1.4.2 Intuitive Explanation

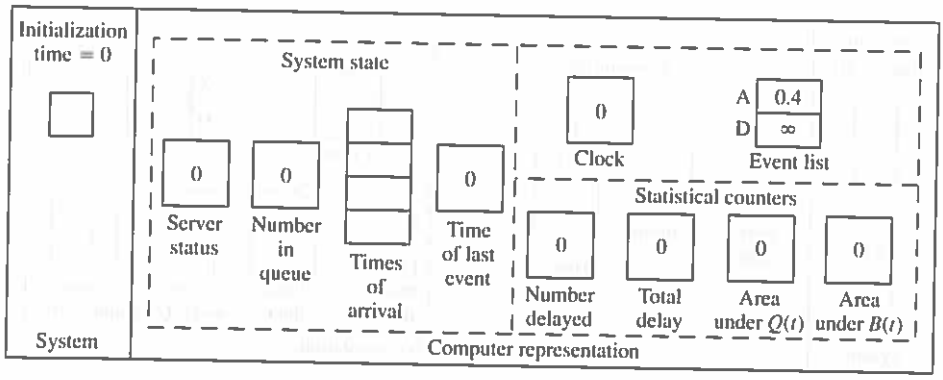
We begin our explanation of how to simulate a single-server queueing system by showing how its simulation model would be represented inside the computer at time $e_0 = 0$ and the times $e_1, e_2, \ldots, e_{13}$ at which the 13 successive events occur that are needed to observe the desired number, $n = 6$, of delays in queue. For expository convenience, we assume that the interarrival and service times of customers are

$$A_1 = 0.4, A_2 = 1.2, A_3 = 0.5, A_4 = 1.7, A_5 = 0.2,$$
$$A_6 = 1.6, A_7 = 0.2, A_8 = 1.4, A_9 = 1.9, \ldots$$
$$S_1 = 2.0, S_2 = 0.7, S_3 = 0.2, S_4 = 1.1, S_5 = 3.7, S_6 = 0.6, \ldots$$

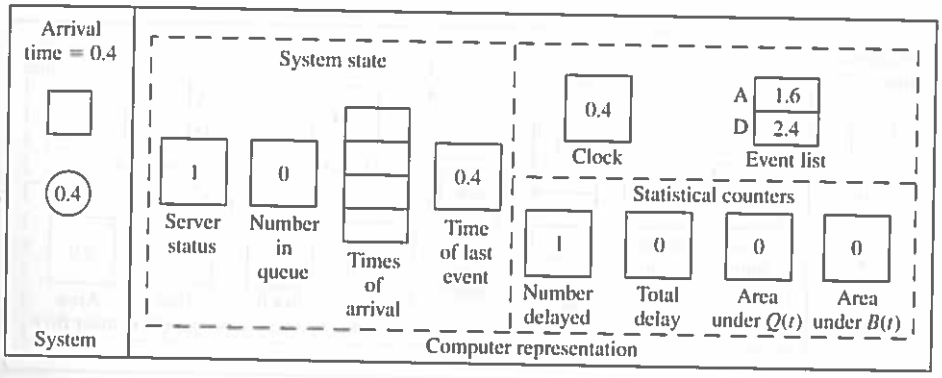
Thus, between time 0 and the time of the first arrival there is 0.4 time unit, between the arrivals of the first and second customers there are 1.2 time units, etc., and the service time required for the first customer is 2.0 time units, etc. Note that it is not necessary to declare what the time units are (minutes, hours, etc.), but only to be sure that all time quantities are expressed in the *same* units. In an actual simulation (see Sec. 1.4.4), the A_i's and the S_i's would be generated from their corresponding probability distributions, as needed, during the course of the simulation. The numerical values for the A_i's and the S_i's given above have been artificially chosen so as to generate the same simulation realization as depicted in Figs. 1.5 and 1.6 illustrating the $Q(t)$ and $B(t)$ processes.

Figure 1.7 gives a snapshot of the system itself and of a computer representation of the system at each of the times $e_0 = 0, e_1 = 0.4, \ldots, e_{13} = 8.6$. In the "system" pictures, the square represents the server, and circles represent customers; the numbers inside the customer circles are the times of their arrivals. In the "computer representation" pictures, the values of the variables shown are *after* all processing has been completed at that event. Our discussion will focus on how the computer representation changes at the event times.

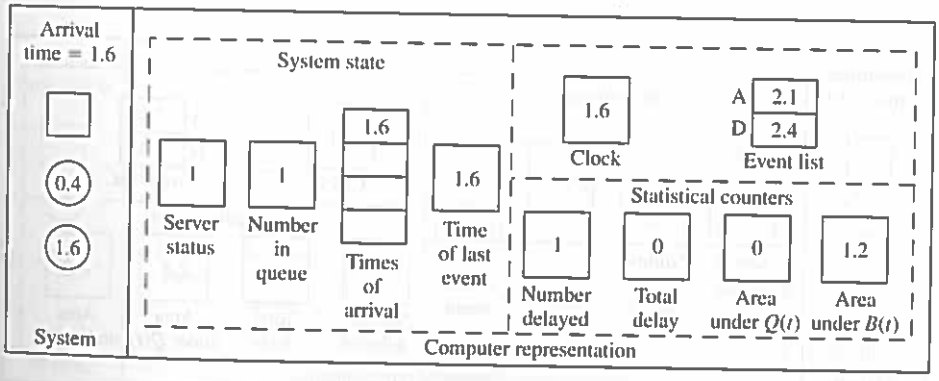
$t = 0$: *Initialization.* The simulation begins with the main program invoking the initialization routine. Our modeling assumption was that initially the system is empty of customers and the server is idle, as depicted in the "system" picture of Fig. 1.7*a*. The model state variables are initialized to represent this: Server status is 0 [we use 0 to represent an idle server and 1 to represent a busy server, similar to the definition of the $B(t)$ function], and the number of customers in the queue is 0. There is a one-dimensional array to store the times of arrival of customers *currently in the queue*; this array is initially empty, and as the simulation progresses, its length will grow and shrink. The time of the last (most recent) event is initialized to 0, so that at the time of the first event (when it is used), it will have its correct value. The simulation



(*a*)

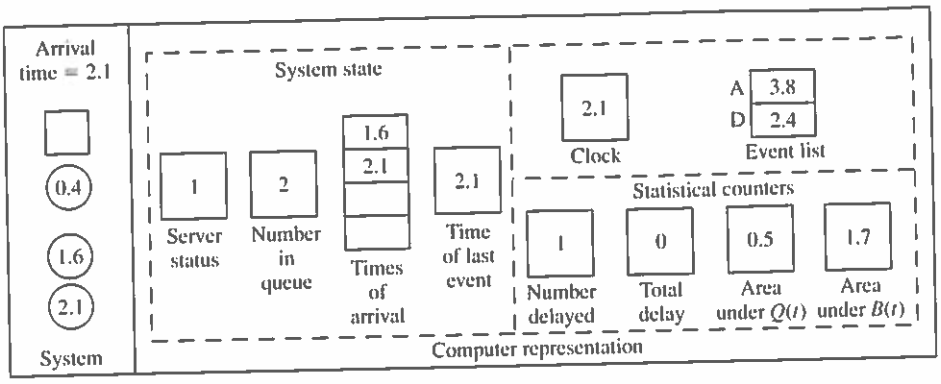


(*b*)

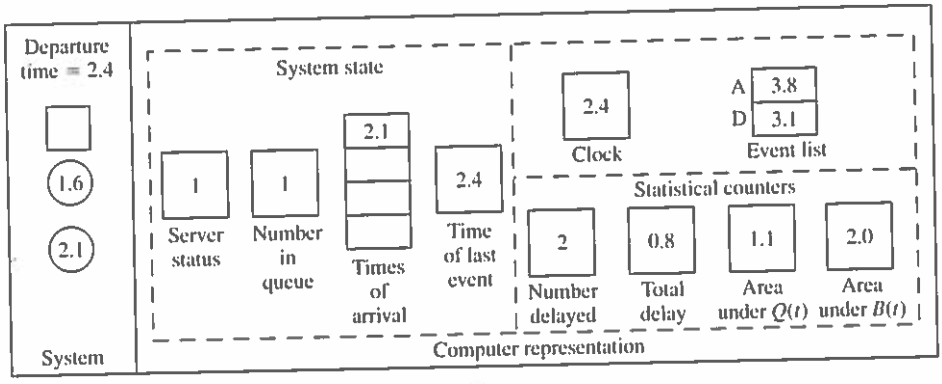


(*c*)

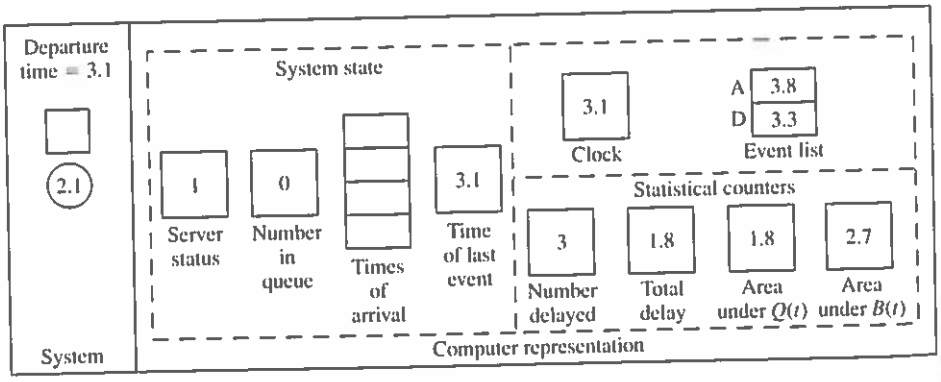
FIGURE 1.7
Snapshots of the system and of its computer representation at time 0 and at each of the 13 succeeding event times.



(*d*)

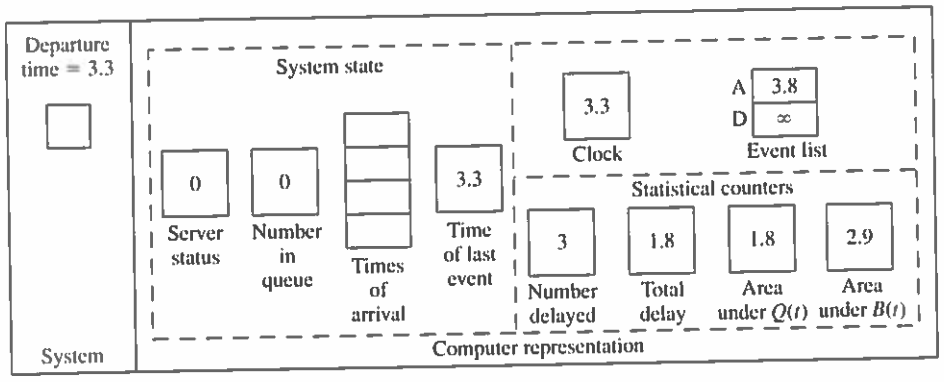


(*e*)

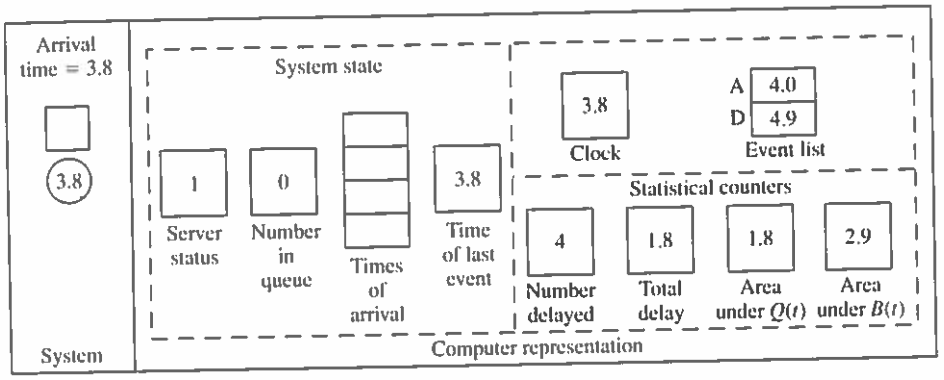


(*f*)

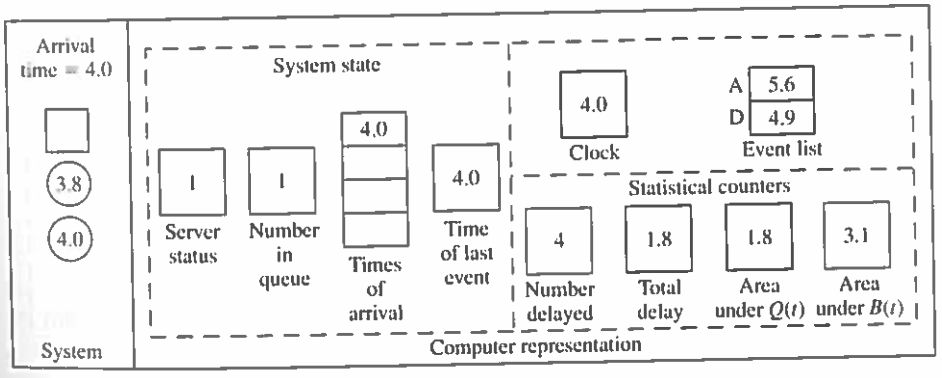
FIGURE 1.7
(*continued*)



(*g*)

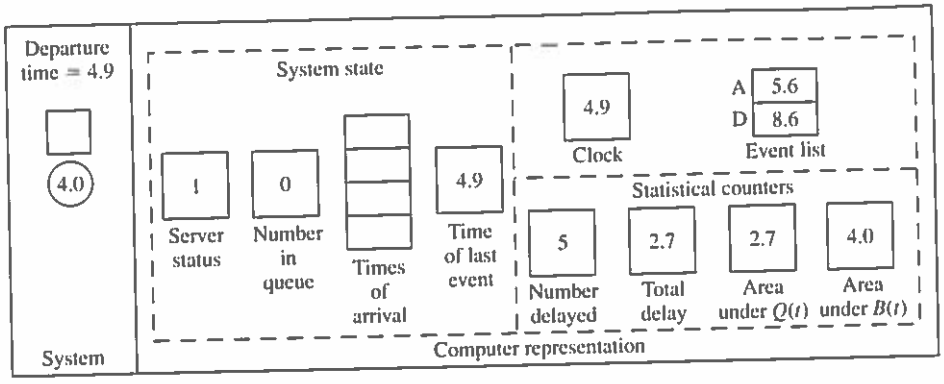


(*h*)

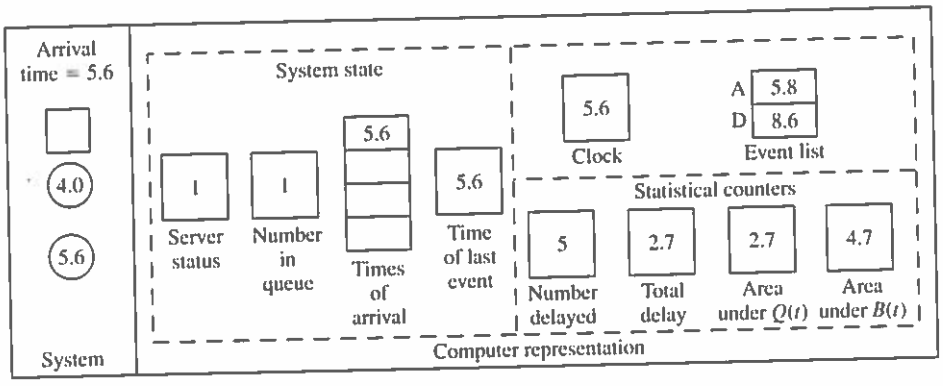


(*i*)

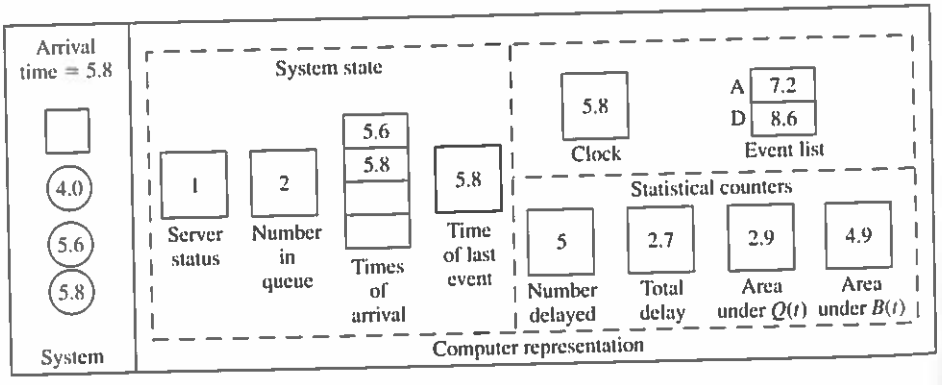
FIGURE 1.7
(*continued*)



(*j*)

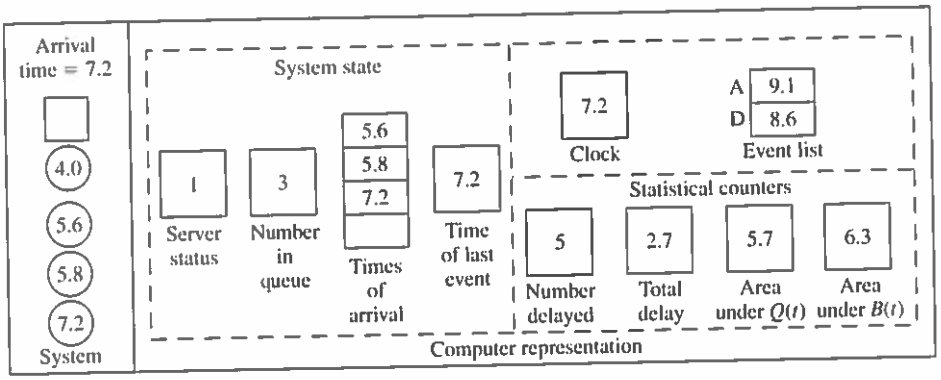


(*k*)

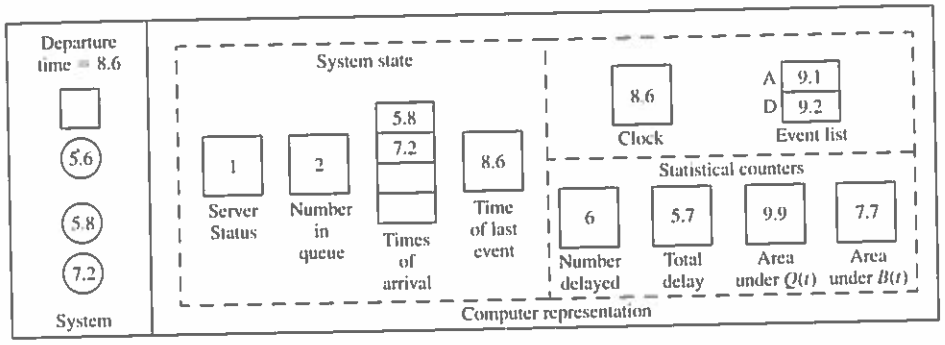


(*l*)

FIGURE 1.7
(*continued*)



(*n*)



(*n*)

FIGURE 1.7
(*continued*)

clock is set to 0, and the *event list*, giving the times of the next occurrence of each of the event types, is initialized as follows. The time of the first arrival is $0 + A_1 = 0.4$, and is denoted by "A" next to the event list. Since there is no customer in service, it does not even make sense to talk about the time of the next departure ("D" by the event list), and we know that the first event will be the initial customer arrival at time 0.4. However, the simulation progresses in general by looking at the event list and picking the smallest value from it to determine what the next event will be, so by scheduling the next departure to occur at time ∞ (or a very large number in a computer program), we effectively eliminate the departure event from consideration and force the next event to be an arrival. Finally, the four statistical counters are initialized to 0. When all initialization is done, control is returned to the main program, which then calls the timing routine to determine the next event. Since $0.4 < \infty$, the next event will be an arrival at time 0.4, and the

timing routine advances the clock to this time, then passes control back to the main program with the information that the next event is to be an arrival.

$t = 0.4$: *Arrival of customer 1.* At time 0.4, the main program passes control to the arrival routine to process the arrival of the first customer. Figure 1.7*b* shows the system and its computer representation *after* all changes have been made to process this arrival. Since this customer arrived to find the server idle (status equal to 0), he begins service immediately and has a delay in queue of $D_1 = 0$ (which *does* count as a delay). The server status is set to 1 to represent that the server is now busy, but the queue itself is still empty. The clock has been advanced to the current time, 0.4, and the event list is updated to reflect this customer's arrival: The next arrival will be $A_2 = 1.2$ time units from now, at time $0.4 + 1.2 = 1.6$, and the next departure (the service completion of the customer now arriving) will be $S_1 = 2.0$ time units from now, at time $0.4 + 2.0 = 2.4$. The number delayed is incremented to 1 (when this reaches $n = 6$, the simulation will end), and $D_1 = 0$ is added into the total delay (still at zero). The area under $Q(t)$ is updated by adding in the product of the *previous* value (i.e., the level it had between the last event and now) of $Q(t)$ (0 in this case) times the width of the interval of time from the last event to now, $t -$ (time of last event) $= 0.4 - 0$ in this case. Note that the time of the last event used here is its *old* value (0), before it is updated to its new value (0.4) in this event routine. Similarly, the area under $B(t)$ is updated by adding in the product of its previous value (0) times the width of the interval of time since the last event. [Look back at Figs. 1.5 and 1.6 to trace the accumulation of the areas under $Q(t)$ and $B(t)$.] Finally, the time of the last event is brought up to the current time, 0.4, and control is passed back to the main program. It invokes the timing routine, which scans the event list for the smallest value, and determines that the next event will be another arrival at time 1.6; it updates the clock to this value and passes control back to the main program with the information that the next event is an arrival.

$t = 1.6$: *Arrival of customer 2.* At this time we again enter the arrival routine, and Fig. 1.7*c* shows the system and its computer representation after all changes have been made to process this event. Since this customer arrives to find the server busy (status equal to 1 upon her arrival), she must queue up in the first location in the queue, her time of arrival is stored in the first location in the array, and the number-in-queue variable rises to 1. The time of the next arrival in the event list is updated to $A_3 = 0.5$ time unit from now, $1.6 + 0.5 = 2.1$; the time of the next departure is not changed, since its value of 2.4 is the departure time of customer 1, who is still in service at this time. Since we are not observing the end of anyone's delay in queue, the number-delayed and total-delay variables are unchanged. The area under $Q(t)$ is increased by 0 [the previous value of $Q(t)$] times the time since the last event, $1.6 - 0.4 = 1.2$. The area under $B(t)$ is increased by 1 [the previous value of $B(t)$] times this same interval of time, 1.2. After updating the

time of the last event to now, control is passed back to the main program and then to the timing routine, which determines that the next event will be an arrival at time 2.1.

$t = 2.1$: *Arrival of customer 3.* Once again the arrival routine is invoked, as depicted in Fig. 1.7*d*. The server stays busy, and the queue grows by one customer, whose time of arrival is stored in the queue array's second location. The next arrival is updated to $t + A_4 = 2.1 + 1.7 = 3.8$, and the next departure is still the same, as we are still waiting for the service completion of customer 1. The delay counters are unchanged, since this is not the end of anyone's delay in queue, and the two area accumulators are updated by adding in 1 [the previous values of both $Q(t)$ and $B(t)$] times the time since the last event, $2.1 - 1.6 = 0.5$. After bringing the time of the last event up to the present, we go back to the main program and invoke the timing routine, which looks at the event list to determine that the next event will be a departure at time 2.4, and updates the clock to that time.

$t = 2.4$: *Departure of customer 1.* Now the main program invokes the departure routine, and Fig. 1.7*e* shows the system and its representation after this occurs. The server will maintain its busy status, since customer 2 moves out of the first place in queue and into service. The queue shrinks by 1, and the time-of-arrival array is moved up one place, to represent that customer 3 is now first in line. Customer 2, now entering service, will require $S_2 = 0.7$ time unit, so the time of the next departure (that of customer 2) in the event list is updated to S_2 time units from now, or to time $2.4 + 0.7 = 3.1$; the time of the next arrival (that of customer 4) is unchanged, since this was scheduled earlier at the time of customer 3's arrival, and we are still waiting at this time for customer 4 to arrive. The delay statistics are updated, since at this time customer 2 is entering service and is completing her delay in queue. Here we make use of the time-of-arrival array, and compute the second delay as the current time minus the second customer's time of arrival, or $D_2 = 2.4 - 1.6 = 0.8$. (Note that the value of 1.6 was stored in the first location in the time-of-arrival array *before* it was changed, so this delay computation would have to be done before advancing the times of arrival in the array.) The area statistics are updated by adding in $2 \times (2.4 - 2.1)$ for $Q(t)$ [note that the previous value of $Q(t)$ was used], and $1 \times (2.4 - 2.1)$ for $B(t)$. The time of the last event is updated, we return to the main program, and the timing routine determines that the next event is a departure at time 3.1.

$t = 3.1$: *Departure of customer 2.* The changes at this departure are similar to those at the departure of customer 1 at time 2.4 just discussed. Note that we observe another delay in queue, and that after this event is processed the queue is again empty, but the server is still busy.

$t = 3.3$: *Departure of customer 3.* Again, the changes are similar to those in the above two departure events, with one important exception: Since the queue is now empty, the server becomes idle and we must set the next departure time in the event list to ∞, since the system now looks the

same as it did at time 0 and we want to force the next event to be the arrival of customer 4.

$t = 3.8$: *Arrival of customer 4.* Since this customer arrives to find the server idle, he has a delay of 0 (i.e., $D_4 = 0$) and goes right into service. Thus, the changes here are very similar to those at the arrival of the first customer at time $t = 0.4$.

The remaining six event times are depicted in Fig. 1.7*i* through 1.7*n*, and readers should work through these to be sure they understand why the variables and arrays are as they appear; it may be helpful to follow along in the plots of $Q(t)$ and $B(t)$ in Figs. 1.5 and 1.6. With the departure of customer 5 at time $t = 8.6$, customer 6 leaves the queue and enters service, at which time the number delayed reaches 6 (the specified value of n) and the simulation ends. At this point, the main program invokes the report generator to compute the final output measures [$\hat{d}(6) = 5.7/6 = 0.95$, $\hat{q}(6) = 9.9/8.6 = 1.15$, and $\hat{u}(6) = 7.7/8.6 = 0.90$] and write them out.

A few specific comments about the above example illustrating the logic of a simulation should be made:

- Perhaps the key element in the dynamics of a simulation is the interaction between the simulation clock and the event list. The event list is maintained, and the clock jumps to the next event, as determined by scanning the event list at the end of each event's processing for the smallest (i.e., next) event time. This is how the simulation progresses through time.
- While processing an event, no "simulated" time passes. However, even though time is standing still for the model, care must be taken to process updates of the state variables and statistical counters in the appropriate order. For example, it would be incorrect to update the number in queue before updating the area-under-$Q(t)$ counter, since the height of the rectangle to be used is the *previous* value of $Q(t)$ [before the effect of the current event on $Q(t)$ has been implemented]. Similarly, it would be incorrect to update the time of the last event before updating the area accumulators. Yet another type of error would result if the queue list were changed at a departure before the delay of the first customer in queue were computed, since his time of arrival to the system would be lost.
- It is sometimes easy to overlook contingencies that seem out of the ordinary but that nevertheless must be accommodated. For example, it would be easy to forget that a departing customer could leave behind an empty queue, necessitating that the server be idled and the departure event again be eliminated from consideration. Also, termination conditions are often more involved than they might seem at first sight; in the above example, the simulation stopped in what seems to be the "usual" way, after a departure of one customer, allowing another to enter service and contribute the last delay needed, but the simulation *could* actually have ended instead with an arrival event—how?
- In some simulations it can happen that two (or more) entries in the event list are tied for smallest, and a decision rule must be incorporated to break such *time ties* (this happens with the inventory simulation considered later in Sec. 1.5). The tie-breaking rule can affect the results of the simulation, so must be chosen in accordance with

how the system is to be modeled. In many simulations, however, we can ignore the possibility of ties, since the use of continuous random variables may make their occurrence an event with probability 0. In the above model, for example, if the interarrival-time or service-time distribution is continuous, then a time tie in the event list is a probability-zero event (though it could still happen during the computer simulation due to finite accuracy in representation of real numbers).

The above exercise is intended to illustrate the changes and data structures involved in carrying out a discrete-event simulation from the event-scheduling point of view, and contains most of the important ideas needed for more complex simulations of this type. The interarrival and service times used could have been drawn from a random-number table of some sort, constructed to reflect the desired probability distributions; this would result in what might be called a *hand simulation*, which in principle could be carried out to any length. The tedium of doing this should now be clear, so we will next turn to the use of computers (which are not easily bored) to carry out the arithmetic and bookkeeping involved in longer or more complex simulations.

1.4.3 Program Organization and Logic

In this section we set up the necessary ingredients for the C program to simulate the single-server queueing system, which is given in Sec. 1.4.4.

There are several reasons for choosing a general-purpose language such as C, rather than more powerful high-level simulation software, for introducing computer simulation at this point:

- By learning to simulate in a general-purpose language, in which one must pay attention to every detail, there will be a greater understanding of how simulations actually operate, and thus less chance of conceptual errors if a switch is later made to high-level simulation software.
- Despite the fact that there is now very good and powerful simulation software available (see Chap. 3), it is sometimes necessary to write at least parts of complex simulations in a general-purpose language if the specific, detailed logic of complex systems is to be represented faithfully.
- General-purpose languages are widely available, and entire simulations are sometimes still written in this way.

It is not our purpose in this book to teach any particular simulation software in detail, although we survey several packages in Chap. 3. With the understanding promoted by our more general approach and by going through our simulations in this and the next chapter, the reader should find it easier to learn a specialized simulation-software product.

The single-server queueing model that we will simulate in the following section differs in two respects from the model used in the previous section:

- The simulation will end when $n = 1000$ delays in queue have been completed, rather than $n = 6$, in order to collect more data (and maybe to impress the reader

with the patience of computers, since we have just slugged it out by hand in the $n = 6$ case in the preceding section). It is important to note that this change in the stopping rule changes the model itself, in that the output measures are defined relative to the stopping rule; hence the presence of the "n" in the notation for the quantities $d(n)$, $q(n)$, and $u(n)$ being estimated.

- The interarrival and service times will now be modeled as independent random variables from exponential distributions with mean 1 minute for the interarrival times and mean 0.5 minute for the service times. The exponential distribution with mean β (any positive real number) is continuous, with probability density function

$$f(x) = \frac{1}{\beta} e^{-x/\beta} \qquad \text{for } x \geq 0$$

(See Chaps. 4 and 6 for more information on density functions in general, and on the exponential distribution in particular.) We make this change here since it is much more common to generate input quantities (which drive the simulation) such as interarrival and service times from specified distributions than to assume that they are "known" as we did in the preceding section. The choice of the exponential distribution with the above particular values of β is essentially arbitrary, and is made primarily because it is easy to generate exponential random variates on a computer. (Actually, the assumption of exponential interarrival times is often quite realistic; assuming exponential service times, however, is less plausible.) Chapter 6 addresses in detail the important issue of how one chooses distribution forms and parameters for modeling simulation input random variables.

The single-server queue with exponential interarrival and service times is commonly called the *M/M/1 queue*, as discussed in App. 1B.

To simulate this model, we need a way to generate random variates from an exponential distribution. First, a *random-number generator* (discussed in detail in Chap. 7) is invoked to generate a variate U that is distributed (continuously) uniformly between 0 and 1; this distribution will henceforth be referred to as U(0, 1) and has probability density function

$$f(x) = \begin{cases} 1 & \text{if } 0 \leq x \leq 1 \\ 0 & \text{otherwise} \end{cases}$$

It is easy to show that the probability that a U(0, 1) random variable falls in any subinterval $[x, x + \Delta x]$ contained in the interval [0, 1] is (uniformly) Δx (see Sec. 6.2.2). The U(0, 1) distribution is fundamental to simulation modeling because, as we shall see in Chap. 8, a random variate from any distribution can be generated by first generating one or more U(0, 1) random variates and then performing some kind of transformation. After obtaining U, we shall take the natural logarithm of it, multiply the result by β, and finally change the sign to return what we will show to be an exponential random variate with mean β, that is, $-\beta \ln U$.

To see why this algorithm works, recall that the (*cumulative*) *distribution function* of a random variable X is defined, for any real x, to be $F(x) = P(X \leq x)$

(Chap. 4 contains a review of basic probability theory). If X is exponential with mean β, then

$$F(x) = \int_0^x \frac{1}{\beta} e^{-t/\beta}\, dt$$
$$= 1 - e^{-x/\beta}$$

for any real $x \geq 0$, since the probability density function of the exponential distribution at the argument $t \geq 0$ is $(1/\beta)e^{-t/\beta}$. To show that our method is correct, we can try to verify that the value it returns will be less than or equal to x (any nonnegative real number), with probability $F(x)$ given above:

$$P(-\beta \ln U \leq x) = P\left(\ln U \geq -\frac{x}{\beta}\right)$$
$$= P(U \geq e^{-x/\beta})$$
$$= P(e^{-x/\beta} \leq U \leq 1)$$
$$= 1 - e^{-x/\beta}$$

The first line in the above is obtained by dividing through by $-\beta$ (recall that $\beta > 0$, so $-\beta < 0$ and the inequality reverses), the second line is obtained by exponentiating both sides (the exponential function is monotone increasing, so the inequality is preserved), the third line is just rewriting, together with knowing that U is in [0, 1] anyway, and the last line follows since U is U(0, 1), and the interval $[e^{-x/\beta}, 1]$ is contained within the interval [0, 1]. Since the last line is $F(x)$ for the exponential distribution, we have verified that our algorithm is correct. Chapter 8 discusses how to generate random variates and processes in general.

In our program, we will use a particular method for random-number generation to obtain the variate U described above, as expressed in the C code of Figs. 7.5 and 7.6 in App. 7A of Chap. 7. While most compilers do have some kind of built-in random-number generator, many of these are of extremely poor quality and should not be used; this issue is discussed fully in Chap. 7.

It is convenient (if not the most computationally efficient) to modularize the programs into several subprograms to clarify the logic and interactions, as discussed in general in Sec. 1.3.2. In addition to a main program, the simulation program includes routines for initialization, timing, report generation, and generating exponential random variates, as in Fig. 1.3. It also simplifies matters if we write a separate routine to update the continuous-time statistics, being the accumulated areas under the $Q(t)$ and $B(t)$ curves. The most important action, however, takes place in the routines for the events, which we number as follows:

Event description	Event type
Arrival of a customer to the system	1
Departure of a customer from the system after completing service	2

Figure 1.8 contains a flowchart for the arrival event. First, the time of the next arrival in the future is generated and placed in the event list. Then a check is made

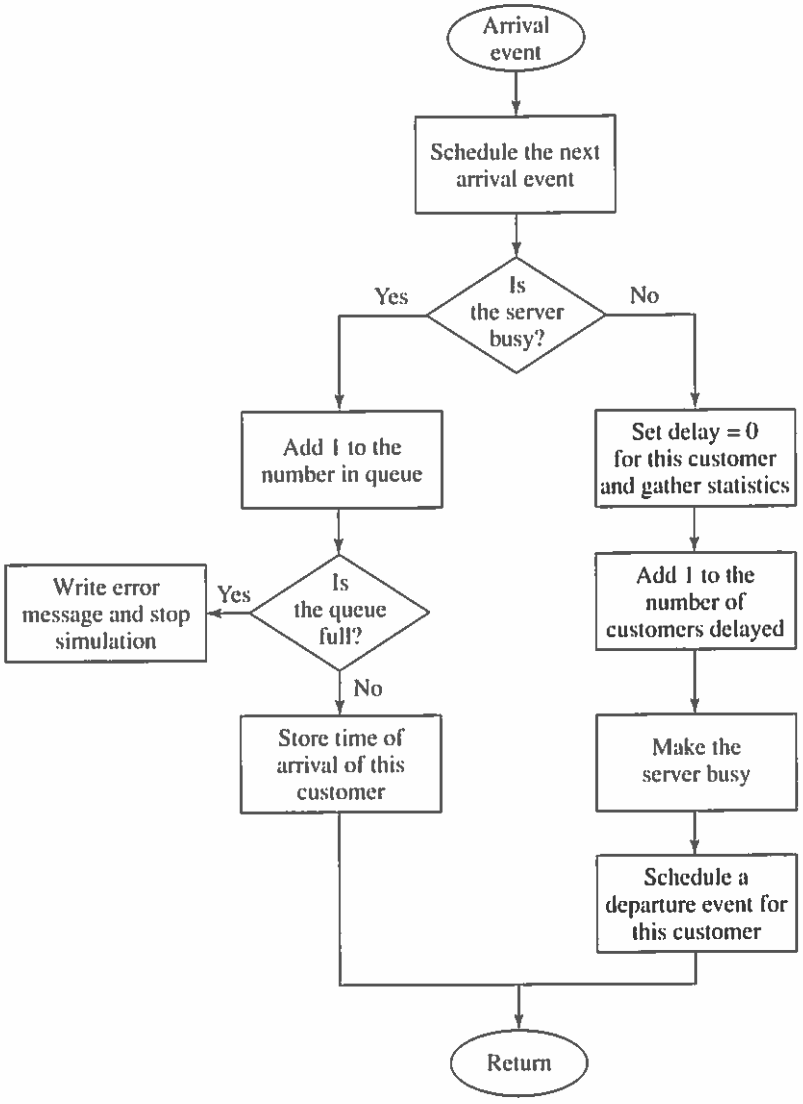


FIGURE 1.8
Flowchart for arrival routine, queueing model.

to determine whether the server is busy. If so, the number of customers in the queue is incremented by 1, and we ask whether the storage space allocated to hold the queue is already full (see the code in Sec. 1.4.4). If the queue is already full, an error message is produced and the simulation is stopped; if there is still room in the queue, the arriving customer's time of arrival is put at the (new) end of the queue. (This queue-full check could be eliminated if using dynamic storage allocation in a programming language that supports this.) On the other hand, if the arriving customer finds the server idle, then this customer has a delay of 0, which *is* counted as a delay, and the number of customer delays completed is incremented by 1. The

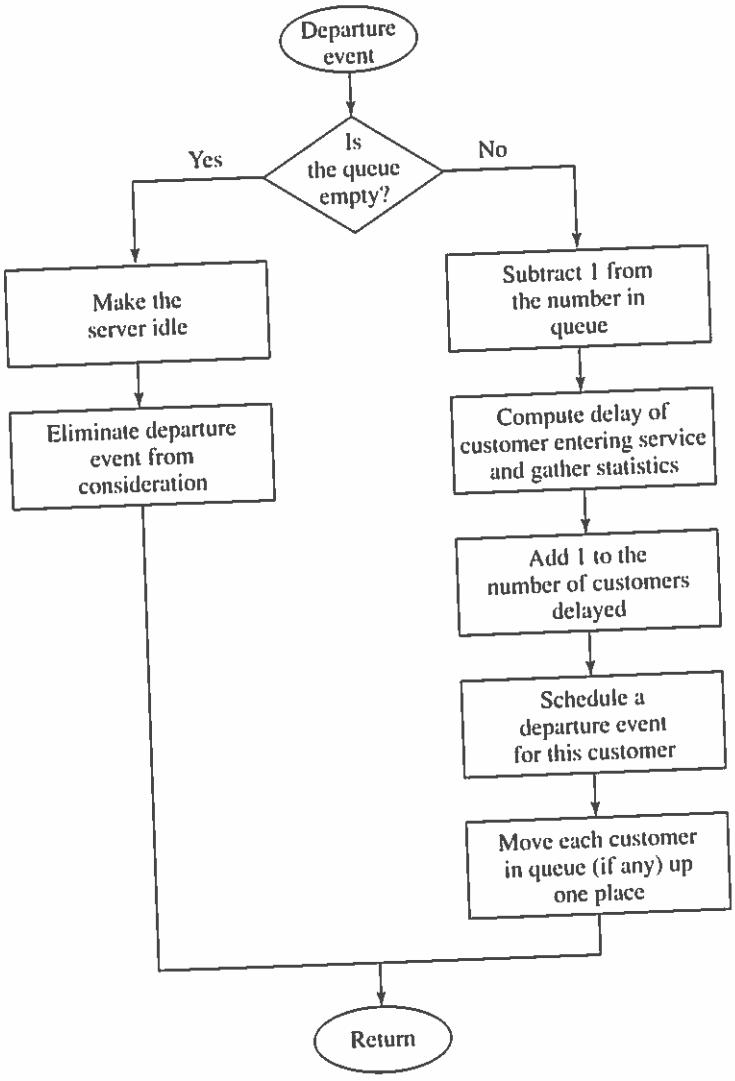


FIGURE 1.9
Flowchart for departure routine, queueing model.

server must be made busy, and the time of departure from service of the arriving customer is scheduled into the event list.

The departure event's logic is depicted in the flowchart of Fig. 1.9. Recall that this routine is invoked when a service completion (and subsequent departure) occurs. If the departing customer leaves no other customers behind in queue, the server is idled and the departure event is eliminated from consideration, since the next event must be an arrival. On the other hand, if one or more customers are left behind by the departing customer, the first customer in queue will leave the queue and enter service, so the queue length is reduced by 1, and the delay in queue of this customer is computed and registered in the appropriate statistical counter. The number

delayed is increased by 1, and a departure event for the customer now entering service is scheduled. Finally, the rest of the queue (if any) is advanced one place. Our implementation of the list for the queue will be very simple in this chapter, and is certainly not the most efficient; Chap. 2 discusses better ways of handling lists to model such things as queues.

In the next section we give an example of how the above setup can be used to write a program in C. The results are discussed in Sec. 1.4.5. This program is neither the simplest nor the most efficient possible, but was instead designed to illustrate how one might organize a program for more complex simulations.

1.4.4 C Program

This section presents a C program for the *M/M/*1 queue simulation. We use the ANSI-standard version of the language, as defined by Kernighan and Ritchie (1988), and in particular use function prototyping. We have also taken advantage of C's facility to give variables and functions fairly long names, which thus should be self-explanatory. (For instance, the current value of simulated time is in a variable called sim_time.) We have run our C program on several different computers and compilers. The numerical results differed in some cases due to inaccuracies in floating-point operations. This can matter if, e.g., at some point in the simulation two events are scheduled very close together in time, and roundoff error results in a different sequencing of the event's occurrences. The C math library must be linked, which might require setting an option depending on the compiler. All code is available at www.mhhe.com/law.

The external definitions are given in Fig. 1.10. The header file lcgrand.h (listed in Fig. 7.6) is included to declare the functions for the random-number generator.

```
/* External definitions for single-server queueing system. */

#include <stdio.h>
#include <math.h>
#include "lcgrand.h"  /* Header file for random-number generator. */

#define Q_LIMIT 100  /* Limit on queue length. */
#define BUSY      1  /* Mnemonics for server's being busy */
#define IDLE      0  /* and idle. */

int   next_event_type, num_custs_delayed, num_delays_required, num_events,
      num_in_q, server_status;
float area_num_in_q, area_server_status, mean_interarrival, mean_service,
      sim_time, time_arrival[Q_LIMIT + 1], time_last_event, time_next_event[3],
      total_of_delays;
FILE  *infile, *outfile;

void  initialize(void);
void  timing(void);
void  arrive(void);
void  depart(void);
void  report(void);
void  update_time_avg_stats(void);
float expon(float mean);
```

FIGURE 1.10
C code for the external definitions, queueing model.

might leave a customer with hair partially cut. In such a case, we might want to close the door of the barbershop after 8 hours but continue to run the simulation until all customers present when the door closes (if any) have been served. The reader is asked in Prob. 1.10 to supply the program changes necessary to implement this stopping rule (see also Sec. 2.6).

1.4.7 Determining the Events and Variables

We defined an event in Sec. 1.3 as an instantaneous occurrence that may change the system state, and in the simple single-server queue of Sec. 1.4.1 it was not too hard to identify the events. However, the question sometimes arises, especially for complex systems, of how one determines the number and definition of events in general for a model. It may also be difficult to specify the state variables needed to keep the simulation running in the correct event sequence and to obtain the desired output measures. There is no completely general way to answer these questions, and different people may come up with different ways of representing a model in terms of events and variables, all of which may be correct. But there are some principles and techniques to help simplify the model's structure and to avoid logical errors.

Schruben (1983b) presented an *event-graph* method, which was subsequently refined and extended by Sargent (1988) and Som and Sargent (1989). In this approach proposed events, each represented by a *node*, are connected by *directed arcs* (arrows) depicting how events may be scheduled from other events and from themselves. For example, in the queueing simulation of Sec. 1.4.3, the arrival event schedules another future occurrence of itself and (possibly) a departure event, and the departure event may schedule another future occurrence of itself; in addition, the arrival event must be initially scheduled in order to get the simulation going. Event graphs connect the proposed set of events (nodes) by arcs indicating the type of event scheduling that can occur. In Fig. 1.25 we show the event graph for our single-server queueing system, where the heavy, smooth arrows indicate that an event at the end of the arrow *may* be scheduled from the event at the beginning of the arrow in a (possibly) *nonzero* amount of time, and the thin jagged arrow indicates that the event at its end is scheduled initially. Thus, the arrival event reschedules itself and may schedule a departure (in the case of an arrival who finds the server idle), and the departure event may reschedule itself (if a departure leaves behind someone else in queue).

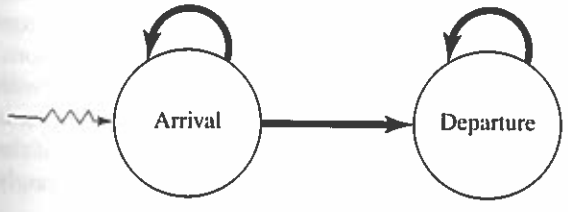


FIGURE 1.25
Event graph, queueing model.

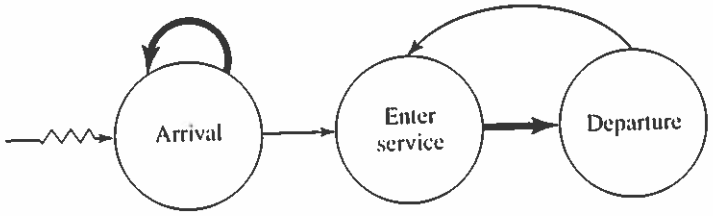


FIGURE 1.26
Event graph, queueing model with separate "enter-service" event.

For this model, it could be asked why we did not explicitly account for the act of a customer's entering service (either from the queue or upon arrival) as a separate event. This certainly can happen, and it could cause the state to change (i.e., the queue length to fall by 1). In fact, this could have been put in as a separate event without making the simulation incorrect, and would give rise to the event diagram in Fig. 1.26. The two thin smooth arrows each represent an event at the beginning of an arrow potentially scheduling an event at the end of the arrow without any intervening time, i.e., immediately; in this case the straight thin smooth arrow refers to a customer who arrives to an empty system and whose "enter-service" event is thus scheduled to occur immediately, and the curved thin smooth arrow represents a customer departing with a queue left behind, and so the first customer in the queue would be scheduled to enter service immediately. The number of events has now increased by 1, and so we have a somewhat more complicated representation of our model. One of the uses of event graphs is to simplify a simulation's event structure by eliminating unnecessary events. There are several "rules" that allow for simplification, and one of them is that if an event node has incoming arcs that are all thin and smooth (i.e., the only way this event is scheduled is by other events and without any intervening time), then this event can be eliminated from the model and its action built into the events that schedule it in zero time. Here, the "enter-service" event could be eliminated, and its action put partly into the arrival event (when a customer arrives to an idle server and begins service immediately) and partly into the departure event (when a customer finishes service and there is a queue from which the next customer is taken to enter service); this takes us back to the simpler event graph in Fig. 1.25. Basically, "events" that can happen only in conjunction with other events do not need to be in the model. Reducing the number of events not only simplifies model conceptualization, but may also speed its execution. Care must be taken, however, when "collapsing" events in this way to handle priorities and time ties appropriately.

Another rule has to do with initialization. The event graph is decomposed into *strongly connected* components, within each of which it is possible to "travel" from every node to every other node by following the arcs in their indicated directions. The graph in Fig. 1.25 decomposes into two strongly connected components (with a single node in each), and that in Fig. 1.26 has two strongly connected components (one of which is the arrival node by itself, and the other of which consists of the enter-service and departure nodes). The initialization rule states that in any strongly

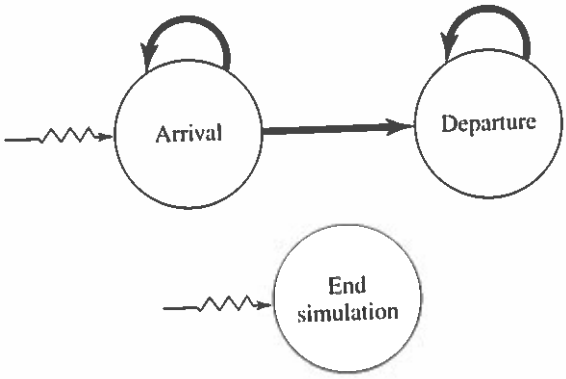


FIGURE 1.27
Event graph, queueing model with fixed run length.

connected component of nodes that has no incoming arcs from other event nodes outside the component, there must be at least one node that is initially scheduled; if this rule were violated, it would never be possible to execute any of the events in the component. In Figs. 1.25 and 1.26, the arrival node is such a strongly connected component since it has no incoming arcs from other nodes, and so it must be initialized. Figure 1.27 shows the event graph for the queueing model of Sec. 1.4.6 with the fixed run length, for which we introduced the dummy "end-simulation" event. Note that this event is itself a strongly connected component without any arcs coming in, and so it must be initialized; i.e., the end of the simulation is scheduled as part of the initialization. Failure to do so would result in erroneous termination of the simulation.

We have presented only a partial and simplified account of the event-graph technique. There are several other features, including event-canceling relations, ways to combine similar events into one, refining the event-scheduling arcs to include conditional scheduling, and incorporating the state variables needed; see the original paper by Schruben (1983b). Sargent (1988) and Som and Sargent (1989) extend and refine the technique, giving comprehensive illustrations involving a flexible manufacturing system and computer network models. Event graphs can also be used to test whether two apparently different models might in fact be equivalent [Yücesan and Schruben (1992)], as well as to forecast how computationally intensive a model will be when it is executed [Yücesan and Schruben (1998)]. Schruben and Schruben (www.sigmawiki.com) developed a software package, SIGMA, for interactive event-graph modeling that runs a model and generates source code. A general event-graph review and tutorial are given by Buss (1996), and advanced applications of event graphs are described in Schruben et al. (2003).

In modeling a system, the event-graph technique can be used to simplify the structure and to detect certain kinds of errors, and is especially useful in complex models involving a large number of interrelated events. Other considerations should also be kept in mind, such as continually asking why a particular state variable is needed; see Prob. 1.4.