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

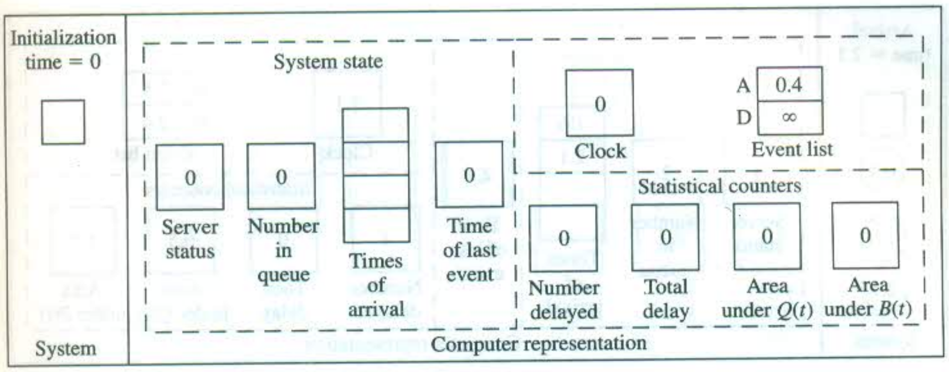
We begin our explanation of how to simulate a single-server queueing system by showing how its simulation model would be represented inside the computer at time $e_0 = 0$ and the times $e_1, e_2, \ldots, e_{13}$ at which the 13 successive events occur that are needed to observe the desired number, $n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

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

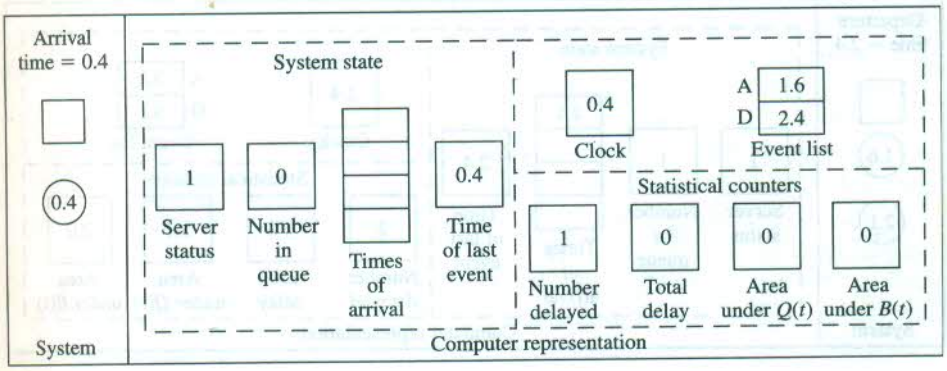
Thus, between time 0 and the time of the first arrival there is 0.4 time unit, between the arrivals of the first and second customers there are 1.2 time units, etc., and the service time required for the first customer is 2.0 time units, etc. Note that it is not necessary to declare what the time units are (minutes, hours, etc.), but only to be sure that all time quantities are expressed in the *same* units. In an actual simulation (see Secs. 1.4.4 and 1.4.5), 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.

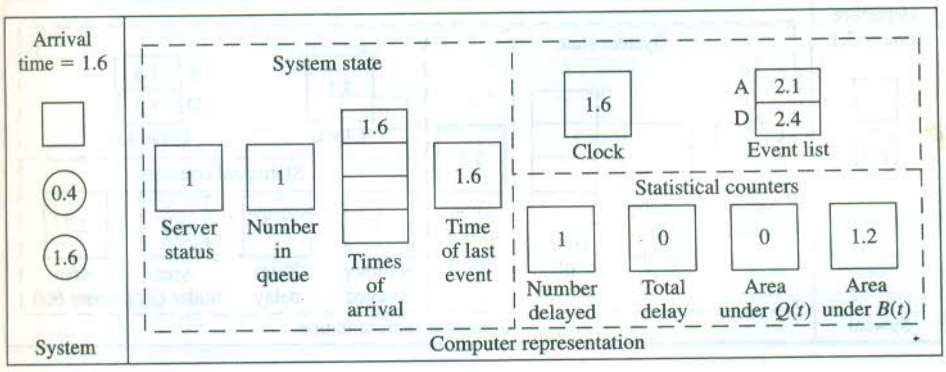
$t = 0$: *Initialization.* The simulation begins with the main program invoking the initialization routine. Our modeling assumption was that initially the system is empty of customers and the server is idle, as depicted in the "system" picture of Fig. 1.7*a*. The model state variables are initialized 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 clock is set to 0, and the *event list*, giving the times of the next



(*a*)

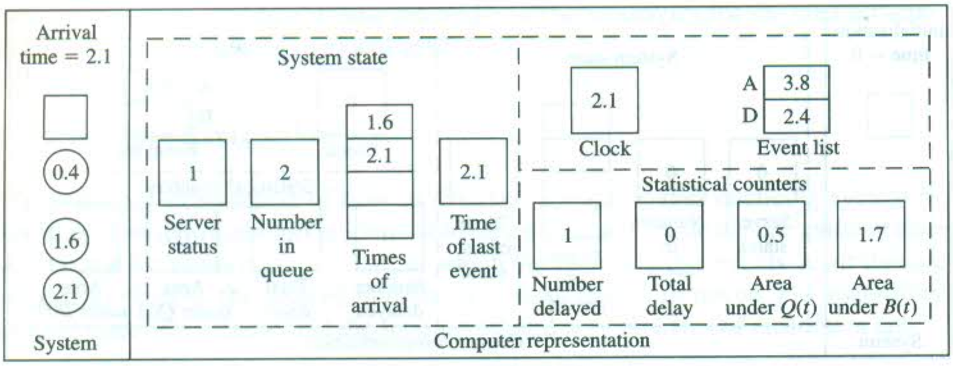


(*b*)

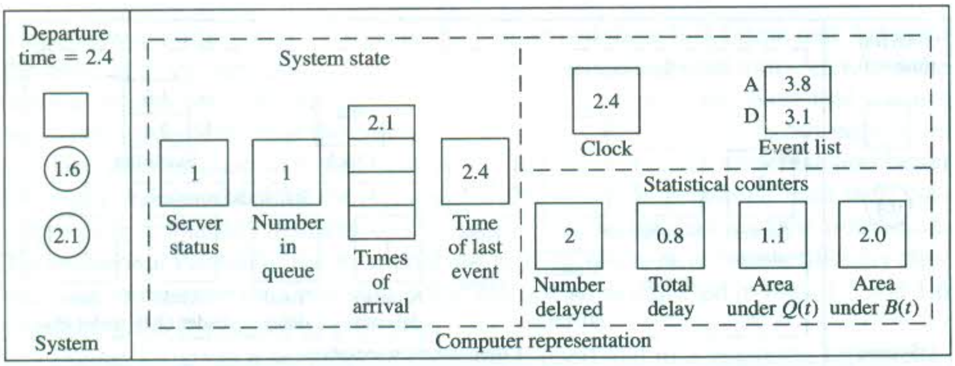


(*c*)

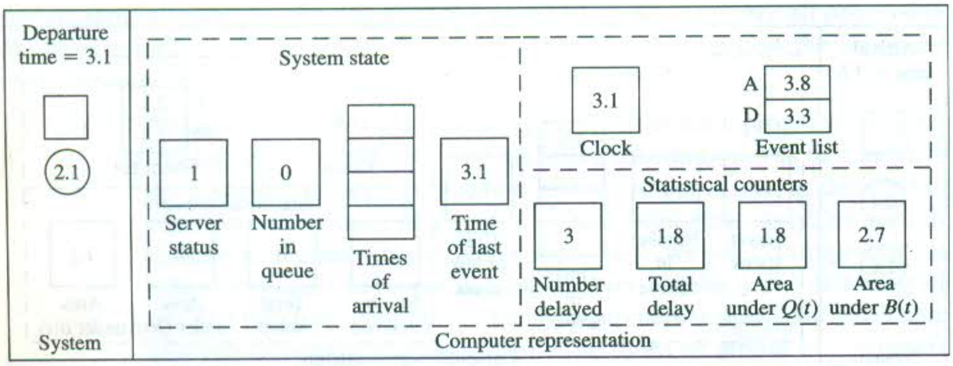
FIGURE 1.7
Snapshots of the system and of its computer representation at time 0 and at each of the 13 succeeding event times.



(*d*)

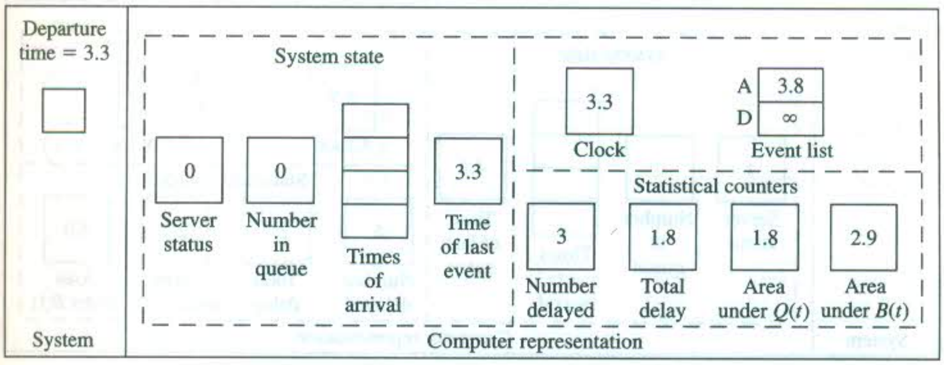


(*e*)

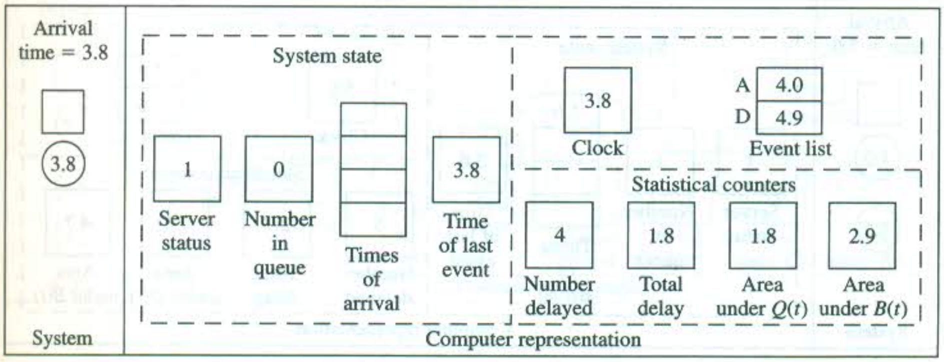


(*f*)

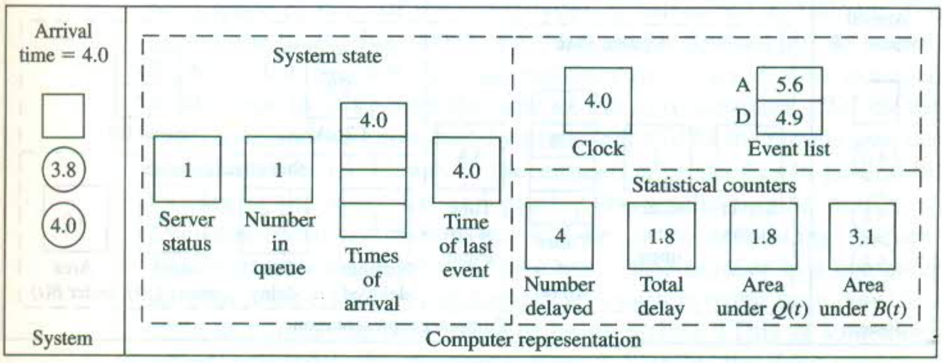
FIGURE 1.7
(*continued*)



(*g*)

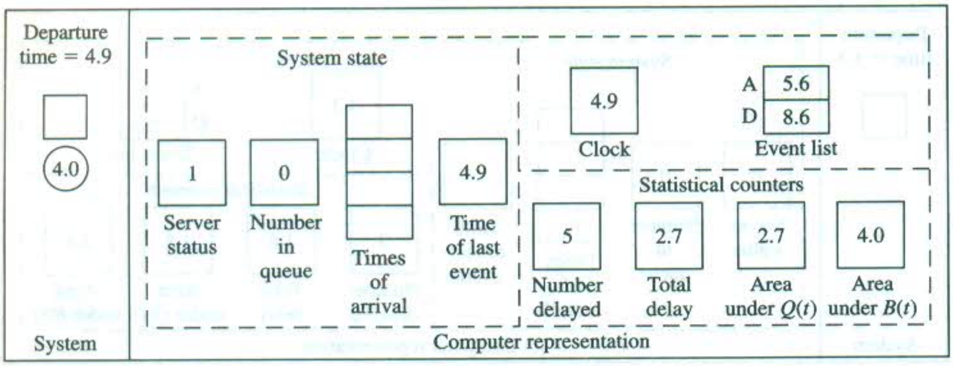


(*h*)

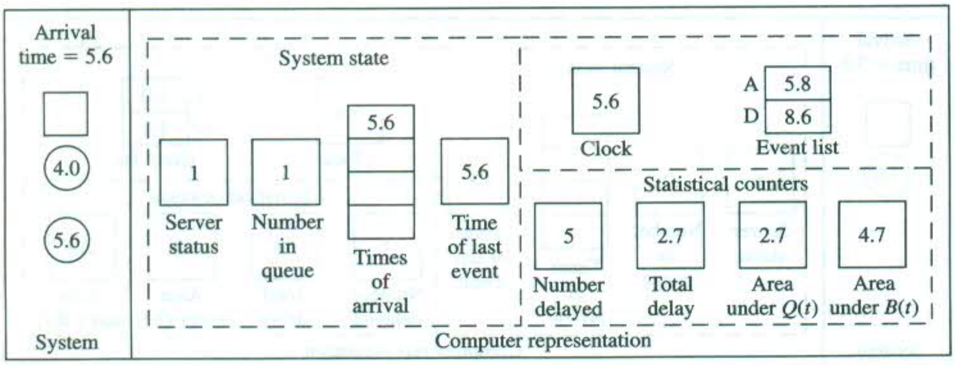


(*i*)

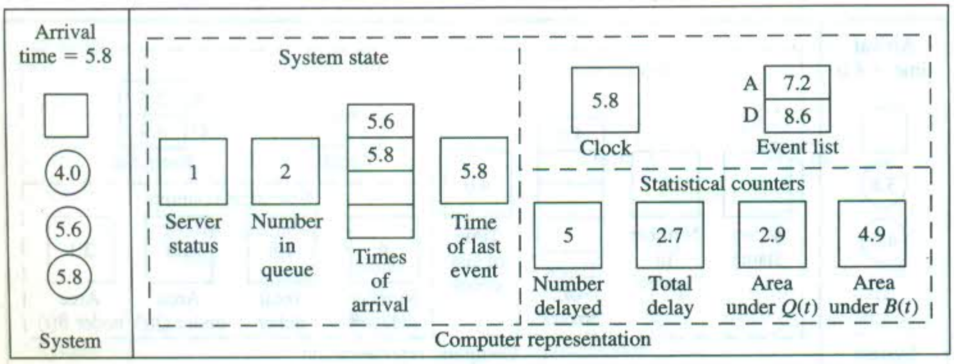
FIGURE 1.7
(*continued*)



(*j*)

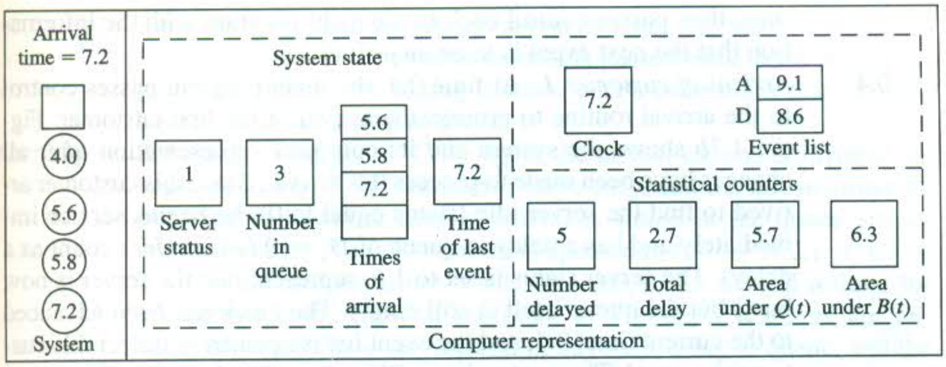


(*k*)

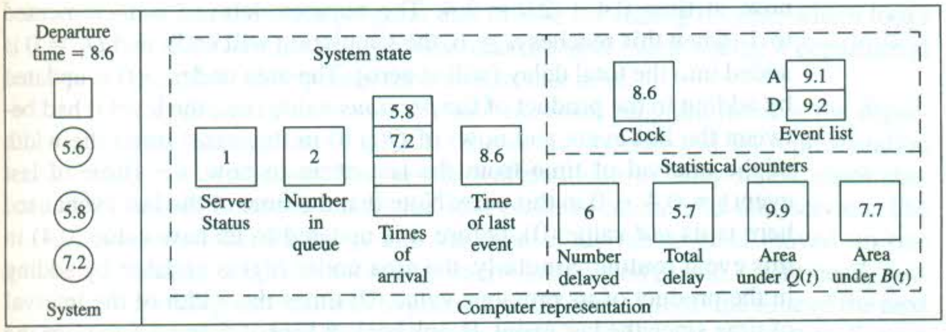


(*l*)

FIGURE 1.7
(*continued*)



(*m*)



(*n*)

FIGURE 1.7
(*continued*)

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. (This is sometimes called *poisoning* the departure event.) 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 Figs. 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 programs to simulate the single-server queueing system in FORTRAN (Sec. 1.4.4) and C (Sec. 1.4.5). The organization and logic described in this section apply for both languages, so the reader need only go through one of Sec. 1.4.4 or 1.4.5, according to language preference.

There are several reasons for choosing a general-purpose language such as FORTRAN or 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 two sections 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

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

  (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. The subprograms used by the FORTRAN and C codes both operate in the same way, which we will now develop. 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

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

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:

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

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 programs, we will use a particular method for random-number generation to obtain the variate U described above, as expressed in the FORTRAN and C codes of Figs. 7.5 through 7.7 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

As the logic of these event routines is independent of the particular language to be used, we shall discuss it here. 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 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 or 1.4.5 for details). 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

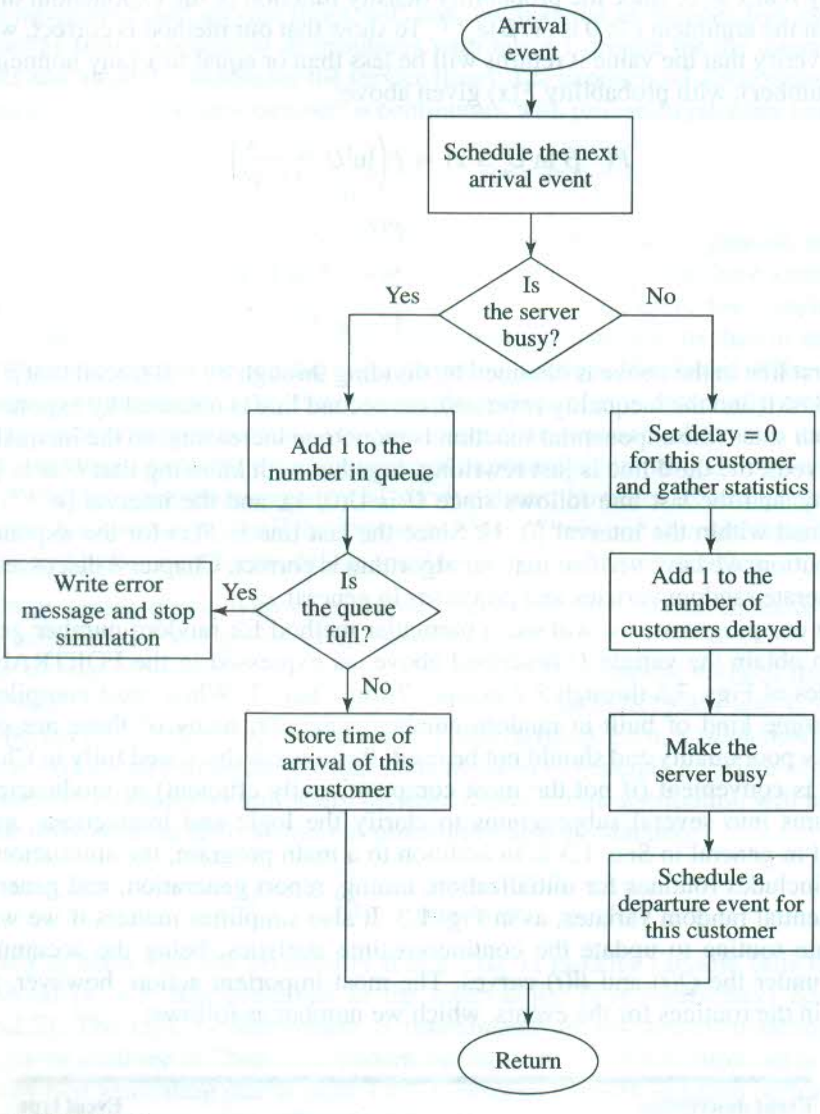

Arrival event
Schedule the next arrival event
Is the server busy?
Yes
No
Add 1 to the number in queue
Set delay = 0 for this customer and gather statistics
Is the queue full?
Yes
Write error message and stop simulation
No
Store time of arrival of this customer
Add 1 to the number of customers delayed
Make the server busy
Schedule a departure event for this customer
Return

FIGURE 1.8
Flowchart for arrival routine, queueing model.

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 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)

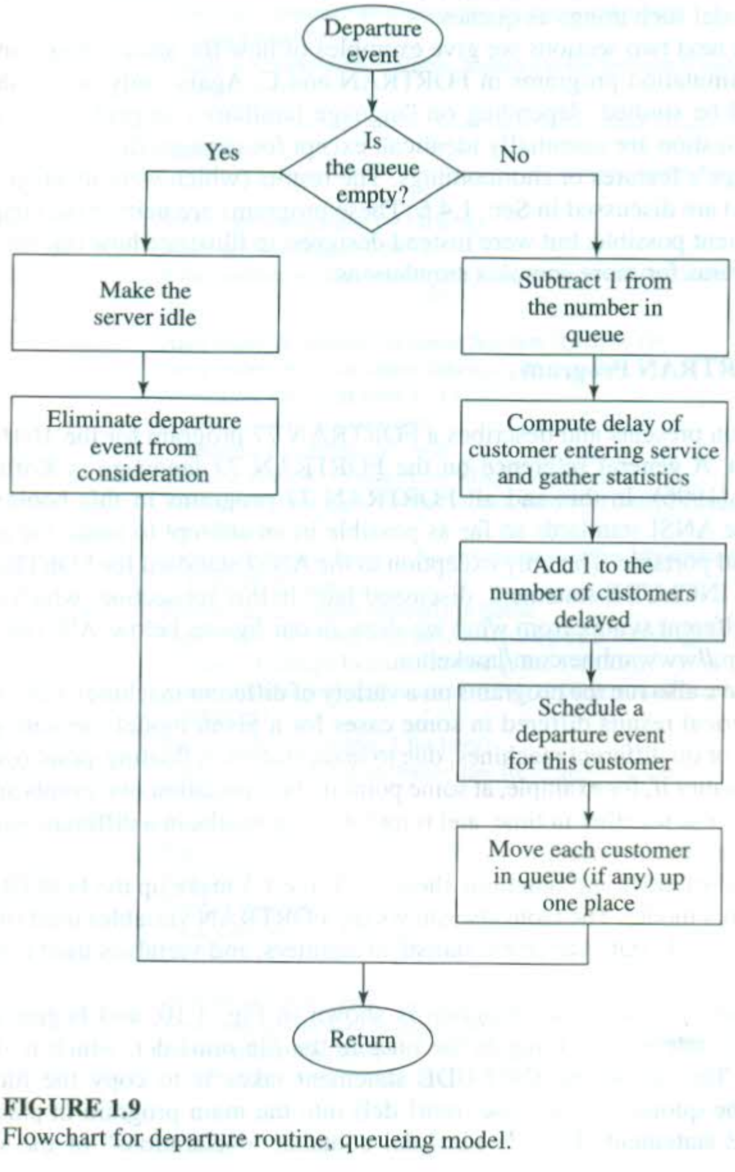

Departure event
Is the queue empty?
Yes
No
Make the server idle
Eliminate departure event from consideration
Subtract 1 from the number in queue
Compute delay of customer entering service and gather statistics
Add 1 to the number of customers delayed
Schedule a departure event for this customer
Move each customer in queue (if any) up one place
Return

FIGURE 1.9
Flowchart for departure routine, queueing model.

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 two sections we give examples of how the above setup can be used to write simulation programs in FORTRAN and C. Again, only one of these sections need be studied, depending on language familiarity or preference; the logic and organization are essentially identical, except for changes dictated by a particular language's features or shortcomings. The results (which were identical for both languages) are discussed in Sec. 1.4.6. These programs are neither the simplest nor most efficient possible, but were instead designed to illustrate how one might organize programs for more complex simulations.

1.4.4 FORTRAN Program

This section presents and describes a FORTRAN 77 program for the *M*/*M*/1 queue simulation. A general reference on the FORTRAN 77 language is Koffman and Friedman (1996). In this and all FORTRAN 77 programs in this book we have obeyed the ANSI standards so far as possible in an attempt to make the programs general and portable. The only exception to the ANSI standard for FORTRAN 77 is use of the INCLUDE statement, discussed later in this subsection, which can have slightly different syntax from what we show in our figures below. All code is available at http://www.mhhe.com/lawkelton.

We have also run the programs on a variety of different machines and compilers. The numerical results differed in some cases for a given model run with different compilers or on different machines, due to inaccuracies in floating-point operations. This can matter if, for example, 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 events' occurrences.

The subroutines and functions shown in Table 1.1 make up the FORTRAN program for this model. The table also shows the FORTRAN variables used (modeling variables include state variables, statistical counters, and variables used to facilitate coding).

The code for the main program is shown in Fig. 1.10, and begins with the INCLUDE statement to bring in the lines in the file mm1.dcl, which is shown in Fig. 1.11. The action the INCLUDE statement takes is to copy the file named between the quotes (in this case, mm1.dcl) into the main program in place of the INCLUDE statement. The file mm1.dcl contains "declarations" of the variables