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.

The symbolic constant Q_LIMIT is set to 100, our guess (which might have to be adjusted by trial and error) as to the longest the queue will ever get. (As mentioned earlier, this guess could be eliminated if we were using dynamic storage allocation; while C supports this, we have not used it in our examples.) The symbolic constants BUSY and IDLE are defined to be used with the server_status variable, for code readability. File pointers *infile and *outfile are defined to allow us to open the input and output files from within the code, rather than at the operating-system level. Note also that the event list, as we have discussed it so far, will be implemented in an array called time_next_event, whose 0th entry will be ignored in order to make the index agree with the event type.

The code for the main function is shown in Fig. 1.11. The input and output files are opened, and the number of event types for the simulation is initialized to 2 for this model. The input parameters then are read in from the file mm1.in, which contains a single line with the numbers 1.0, 0.5, and 1000, separated by blanks. After writing a report heading and echoing the input parameters (as a check that they were read correctly), the initialization function is invoked. The "while" loop then executes the simulation as long as more customer delays are needed to fulfill the 1000-delay stopping rule. Inside the "while" loop, the timing function is first invoked to determine the type of the next event to occur and to advance the simulation clock to its time. Before processing this event, the function to update the areas under the $Q(t)$ and $B(t)$ curves is invoked; by doing this at this time we automatically update these areas before processing each event. Then a switch statement, based on next_event_type (=1 for an arrival and 2 for a departure), passes control to the appropriate event function. After the "while" loop is done, the report function is invoked, the input and output files are closed, and the simulation ends.

Code for the initialization function is given in Fig. 1.12. Each statement here corresponds to an element of the computer representation in Fig. 1.7*a*. Note that the time of the first arrival, time_next_event[1], is determined by adding an exponential random variate with mean mean_interarrival, namely, expon(mean_interarrival), to the simulation clock, sim_time = 0. (We explicitly used "sim_time" in this statement, although it has a value of 0, to show the general form of a statement to determine the time of a future event.) Since no customers are present at time sim_time = 0, the time of the next departure, time_next_event[2], is set to 1.0e + 30 (C notation for 10^{30}), guaranteeing that the first event will be an arrival.

The timing function, which is given in Fig. 1.13, is used to compare time_next_event[1], time_next_event[2], . . . , time_next_event[num_events] (recall that num_events was set in the main function) and to set next_event_type equal to the event type whose time of occurrence is the smallest. In case of ties, the lowest-numbered event type is chosen. Then the simulation clock is advanced to the time of occurrence of the chosen event type, min_time_next_event. The program is complicated slightly by an error check for the event list's being empty, which we define to mean that all events are scheduled to occur at time = 10^{30}. If this is ever the case (as indicated by next_event_type = 0), an error message is produced along with the current clock time (as a possible debugging aid), and the simulation is terminated.

```
main()  /* Main function. */
{
    /* Open input and output files. */

    infile  = fopen("mm1.in",  "r");
    outfile = fopen("mm1.out", "w");

    /* Specify the number of events for the timing function. */

    num_events = 2;

    /* Read input parameters. */

    fscanf(infile, "%f %f %d", &mean_interarrival, &mean_service,
           &num_delays_required);

    /* Write report heading and input parameters. */

    fprintf(outfile, "Single-server queueing system\n\n");
    fprintf(outfile, "Mean interarrival time%11.3f minutes\n\n",
            mean_interarrival);
    fprintf(outfile, "Mean service time%16.3f minutes\n\n", mean_service);
    fprintf(outfile, "Number of customers%14d\n\n", num_delays_required);

    /* Initialize the simulation. */

    initialize();

    /* Run the simulation while more delays are still needed. */

    while (num_custs_delayed < num_delays_required) {

        /* Determine the next event. */

        timing();

        /* Update time-average statistical accumulators. */

        update_time_avg_stats();

        /* Invoke the appropriate event function. */

        switch (next_event_type) {
            case 1:
                arrive();
                break;
            case 2:
                depart();
                break;
        }
    }

    /* Invoke the report generator and end the simulation. */

    report();

    fclose(infile);
    fclose(outfile);

    return 0;
}
```

FIGURE 1.11
C code for the main function, queueing model.

```
void initialize(void)  /* Initialization function. */
{
    /* Initialize the simulation clock. */

    sim_time = 0.0;

    /* Initialize the state variables. */

    server_status   = IDLE;
    num_in_q        = 0;
    time_last_event = 0.0;

    /* Initialize the statistical counters. */

    num_custs_delayed  = 0;
    total_of_delays    = 0.0;
    area_num_in_q      = 0.0;
    area_server_status = 0.0;

    /* Initialize event list.  Since no customers are present, the departure
       (service completion) event is eliminated from consideration. */

    time_next_event[1] = sim_time + expon(mean_interarrival);
    time_next_event[2] = 1.0e+30;
}
```

FIGURE 1.12
C code for function initialize, queueing model.

```
void timing(void)  /* Timing function. */
{
    int   i;
    float min_time_next_event = 1.0e+29;

    next_event_type = 0;

    /* Determine the event type of the next event to occur. */

    for (i = 1; i <= num_events; ++i)
        if (time_next_event[i] < min_time_next_event) {
            min_time_next_event = time_next_event[i];
            next_event_type     = i;
        }

    /* Check to see whether the event list is empty. */

    if (next_event_type == 0) {

        /* The event list is empty, so stop the simulation. */

        fprintf(outfile, "\nEvent list empty at time %f", sim_time);
        exit(1);
    }

    /* The event list is not empty, so advance the simulation clock. */

    sim_time = min_time_next_event;
}
```

FIGURE 1.13
C code for function timing, queueing model.

```
void arrive(void)  /* Arrival event function. */
{
    float delay;

    /* Schedule next arrival. */

    time_next_event[1] = sim_time + expon(mean_interarrival);

    /* Check to see whether server is busy. */

    if (server_status == BUSY) {

        /* Server is busy, so increment number of customers in queue. */

        ++num_in_q;

        /* Check to see whether an overflow condition exists. */

        if (num_in_q > Q_LIMIT) {

            /* The queue has overflowed, so stop the simulation. */

            fprintf(outfile, "\nOverflow of the array time_arrival at");
            fprintf(outfile, " time %f", sim_time);
            exit(2);
        }

        /* There is still room in the queue, so store the time of arrival of the
           arriving customer at the (new) end of time_arrival. */

        time_arrival[num_in_q] = sim_time;
    }

    else {

        /* Server is idle, so arriving customer has a delay of zero.  (The
           following two statements are for program clarity and do not affect
           the results of the simulation.) */

        delay            = 0.0;
        total_of_delays += delay;

        /* Increment the number of customers delayed, and make server busy. */

        ++num_custs_delayed;
        server_status = BUSY;

        /* Schedule a departure (service completion). */

        time_next_event[2] = sim_time + expon(mean_service);
    }
}
```

FIGURE 1.14
C code for function arrive, queueing model.

The code for event function arrive is in Fig. 1.14, and follows the discussion as given in Sec. 1.4.3 and in the flowchart of Fig. 1.8. Note that "sim_time" is the time of arrival of the customer who is just now arriving, and that the queue-overflow check is made by asking whether num_in_q is now greater than Q_LIMIT, the length for which the array time_arrival was dimensioned.

Event function depart, whose code is shown in Fig. 1.15, is invoked from the main program when a service completion (and subsequent departure) occurs; the

```
void depart(void)  /* Departure event function. */
{
    int   i;
    float delay;

    /* Check to see whether the queue is empty. */

    if (num_in_q == 0) {

        /* The queue is empty so make the server idle and eliminate the
           departure (service completion) event from consideration. */

        server_status      = IDLE;
        time_next_event[2] = 1.0e+30;
    }

    else {

        /* The queue is nonempty, so decrement the number of customers in
           queue. */

        --num_in_q;

        /* Compute the delay of the customer who is beginning service and update
           the total delay accumulator. */

        delay            = sim_time - time_arrival[1];
        total_of_delays += delay;

        /* Increment the number of customers delayed, and schedule departure. */

        ++num_custs_delayed;
        time_next_event[2] = sim_time + expon(mean_service);

        /* Move each customer in queue (if any) up one place. */

        for (i = 1; i <= num_in_q; ++i)
            time_arrival[i] = time_arrival[i + 1];
    }
}
```

FIGURE 1.15
C code for function depart, queueing model.

logic for it was discussed in Sec. 1.4.3, with the flowchart in Fig. 1.9. Note that if the statement "time_next_event[2] = 1.0e + 30;" just before the "else" were omitted, the program would get into an infinite loop. (Why?) Advancing the rest of the queue (if any) one place by the "for" loop near the end of the function ensures that the arrival time of the next customer entering service (after being delayed in queue) will always be stored in time_arrival[1]. Note that if the queue were now empty (i.e., the customer who just left the queue and entered service had been the only one in queue), then num_in_q would be equal to 0, and this loop would not be executed at all since the beginning value of the loop index, i, starts out at a value (1) that would already exceed its final value (num_in_q = 0). (Managing the queue in this simple way is certainly inefficient, and could be improved by using pointers; we return to this issue in Chap. 2.) A final comment about depart concerns the subtraction of time_arrival[1] from the clock value, sim_time, to obtain the delay in queue. If the simulation is to run for a long period of (simulated) time, both sim_time and time_arrival[1] would become very large numbers in comparison with the

```
void report(void)  /* Report generator function. */
{
    /* Compute and write estimates of desired measures of performance. */

    fprintf(outfile, "\n\nAverage delay in queue%11.3f minutes\n\n",
            total_of_delays / num_custs_delayed);
    fprintf(outfile, "Average number in queue%10.3f\n\n",
            area_num_in_q / sim_time);
    fprintf(outfile, "Server utilization%15.3f\n\n",
            area_server_status / sim_time);
    fprintf(outfile, "Time simulation ended%12.3f minutes", sim_time);
}
```

FIGURE 1.16
C code for function report, queueing model.

difference between them; thus, since they are both stored as floating-point (float) numbers with finite accuracy, there is potentially a serious loss of precision when doing this subtraction. For this reason, it may be necessary to make both sim_time and the time_arrival array of type double if we are to run this simulation out for a long period of time.

The code for the report function, invoked when the "while" loop in the main program is over, is given in Fig. 1.16. The average delay is computed by dividing the total of the delays by the number of customers whose delays were observed, and the time-average number in queue is obtained by dividing the area under $Q(t)$, now updated to the end of the simulation (since the function to update the areas is called from the main program before processing either an arrival or departure, one of which will end the simulation), by the clock value at termination. The server utilization is computed by dividing the area under $B(t)$ by the final clock time, and all three measures are written out directly. We also write out the final clock value itself, to see how long it took to observe the 1000 delays.

Function update_time_avg_stats is shown in Fig. 1.17. This function is invoked just before processing each event (of any type) and updates the areas under the two functions needed for the continuous-time statistics; this routine is separate for

```
void update_time_avg_stats(void)  /* Update area accumulators for time-average
                                     statistics. */
{
    float time_since_last_event;

    /* Compute time since last event, and update last-event-time marker. */

    time_since_last_event = sim_time - time_last_event;
    time_last_event       = sim_time;

    /* Update area under number-in-queue function. */

    area_num_in_q      += num_in_q * time_since_last_event;

    /* Update area under server-busy indicator function. */

    area_server_status += server_status * time_since_last_event;
}
```

FIGURE 1.17
C code for function update_time_avg_stats, queueing model.

```
float expon(float mean)  /* Exponential variate generation function. */
{
    /* Return an exponential random variate with mean "mean". */

    return -mean * log(lcgrand(1));
}
```

FIGURE 1.18
C code for function expon.

coding convenience only, and is *not* an event routine. The time since the last event is first computed, and then the time of the last event is brought up to the current time in order to be ready for the next entry into this function. Then the area under the number-in-queue function is augmented by the area of the rectangle under $Q(t)$ during the interval since the previous event, which is of width time_since_last_event and of height num_in_q; remember, this function is invoked *before* processing an event, and state variables such as num_in_q still have their previous values. The area under $B(t)$ is then augmented by the area of a rectangle of width time_since_last_event and height server_status; this is why it is convenient to define server_status to be either 0 or 1. Note that this function, like depart, contains a subtraction of two floating-point numbers (sim_time − time_last_event), both of which could become quite large relative to their difference if we were to run the simulation for a long time; in this case it might be necessary to declare both sim_time and time_last_event to be of type double.

The function expon, which generates an exponential random variate with mean β = mean (passed into expon), is shown in Fig. 1.18, and follows the algorithm discussed in Sec. 1.4.3. The random-number generator lcgrand, used here with an int argument of 1, is discussed fully in Chap. 7, and is shown specifically in Fig. 7.5. The C predefined function log returns the natural logarithm of its argument.

The program described here must be combined with the random-number-generator code from Fig. 7.5. This could be done by separate compilations, followed by linking the object codes together in an installation-dependent way.

1.4.5 Simulation Output and Discussion

The output (in a file named mm1.out) is shown in Fig. 1.19. In this run, the average delay in queue was 0.430 minute, there was an average of 0.418 customer in the queue, and the server was busy 46 percent of the time. It took 1027.915 simulated minutes to run the simulation to the completion of 1000 delays, which seems reasonable since the expected time between customer arrivals was 1 minute. (It is not a coincidence that the average delay, average number in queue, and utilization are all so close together for this model; see App. 1B.)

Note that these particular numbers in the output were determined, at root, by the numbers the random-number generator happened to come up with this time. If a different random-number generator were used, or if this one were used in another way (with another "seed" or "stream," as discussed in Chap. 7), then different numbers would have been produced in the output. Thus, these numbers are not to be regarded

```
Single-server queueing system

Mean interarrival time      1.000 minutes

Mean service time           0.500 minutes

Number of customers          1000


Average delay in queue      0.430 minutes

Average number in queue     0.418

Server utilization          0.460

Time simulation ended    1027.915 minutes
```

FIGURE 1.19
Output report, queueing model.

as "The Answers," but rather as estimates (and perhaps poor ones) of the expected quantities we want to know about, $d(n)$, $q(n)$, and $u(n)$; the statistical analysis of simulation output data is discussed in Chaps. 9 through 12. Also, the results are functions of the input parameters, in this case the mean interarrival and service times, and the $n = 1000$ stopping rule; they are also affected by the way we initialized the simulation (empty and idle).

In some simulation studies, we might want to estimate *steady-state* characteristics of the model (see Chap. 9), i.e., characteristics of a model after the simulation has been running a very long (in theory, an infinite amount of) time. For the simple *M*/*M*/1 queue we have been considering, it is possible to compute *analytically* the steady-state average delay in queue, the steady-state time-average number in queue, and the steady-state server utilization, all of these measures of performance being 0.5 [see, e.g., Ross (2003, pp. 480–487)]. Thus, if we wanted to determine these steady-state measures, our estimates based on the stopping rule $n = 1000$ delays were not too far off, at least in absolute terms. However, we were somewhat lucky, since $n = 1000$ was chosen arbitrarily! In practice, the choice of a stopping rule that will give good estimates of steady-state measures is quite difficult. To illustrate this point, suppose for the *M/M*/1 queue that the arrival rate of customers were increased from 1 per minute to 1.98 per minute (the mean interarrival time is now 0.505 minute), that the mean service time is unchanged, and that we wish to estimate the steady-state measures from a run of length $n = 1000$ delays, as before. We performed this simulation run and got values for the average delay, average number in queue, and server utilization of 17.404 minutes, 34.831, and 0.997, respectively. Since the true steady-state values of these measures are 49.5 minutes, 98.01, and 0.99 (respectively), it is clear that the stopping rule cannot be chosen arbitrarily. We discuss how to specify the run length for a steady-state simulation in Chap. 9.

The reader may have wondered why we did not estimate the expected average waiting time in the system of a customer, $w(n)$, rather than the expected average delay in queue, $d(n)$, where the waiting time of a customer is defined as the time interval from the instant the customer arrives to the instant the customer completes service and departs. There were two reasons. First, for many queueing systems we

believe that the customer's delay in queue while waiting for other customers to be served is the most troublesome part of the customer's wait in the system. Moreover, if the queue represents part of a manufacturing system where the "customers" are actually parts waiting for service at a machine (the "server"), then the delay in queue represents a loss, whereas the time spent in service is "necessary." Our second reason for focusing on the delay in queue is one of statistical efficiency. The usual estimator of $w(n)$ would be

$$\hat{w}(n) = \frac{\sum_{i=1}^{n} W_i}{n} = \frac{\sum_{i=1}^{n} D_i}{n} + \frac{\sum_{i=1}^{n} S_i}{n} = \hat{d}(n) + \bar{S}(n) \tag{1.7}$$

where $W_i = D_i + S_i$ is the waiting time in the system of the ith customer and $\bar{S}(n)$ is the average of the n customers' service times. Since the service-time distribution would have to be known to perform a simulation in the first place, the expected or mean service time, $E(S)$, would also be known and an alternative estimator of $w(n)$ is

$$\tilde{w}(n) = \hat{d}(n) + E(S)$$

[Note that $\bar{S}(n)$ is an unbiased estimator of $E(S)$ in Eq. (1.7).] In almost all queueing simulations, $\tilde{w}(n)$ will be a more efficient (less variable) estimator of $w(n)$ than $\hat{w}(n)$ and is thus preferable (both estimators are unbiased). Therefore, if one wants an estimate of $w(n)$, estimate $d(n)$ and add the known expected service time, $E(S)$. In general, the moral is to replace estimators by their expected values whenever possible (see the discussion of indirect estimators in Sec. 11.5).

1.4.6 Alternative Stopping Rules

In the above queueing example, the simulation was terminated when the number of customers delayed became equal to 1000; the final value of the simulation clock was thus a random variable. However, for many real-world models, the simulation is to stop after some fixed amount of time, say 8 hours. Since the interarrival and service times for our example are continuous random variables, the probability of the simulation's terminating after exactly 480 minutes is 0 (neglecting the finite accuracy of a computer). Therefore, to stop the simulation at a specified time, we introduce a dummy "end-simulation" event (call it an event of type 3), which is scheduled to occur at time 480. When the time of occurrence of this event (being held in the third spot of the event list) is less than all other entries in the event list, the report generator is called and the simulation is terminated. The number of customers delayed is now a random variable.

These ideas can be implemented in the program by making changes in the external definitions, the main function, and the initialize and report functions, as shown in Figs. 1.20 through 1.23; the rest of the program is unaltered. In Figs. 1.20 and 1.21, note that we now have three events; that the desired simulation run length, time_end, is now an input parameter (num_delays_required has been removed); and