

SIMULATION PROGRAMMING

ANSI standard C

References: Kernighan & Ritchie and Harbison & Steele (See reading list)

Function prototyping: The use of function declarators, e.g.,

```
int    myfunction ();  
int    myfunction (double, float);  
void   subprogram (void);  
etc.
```

Take advantage of C's facility to give variables and functions relatively **long names**, e.g., `update_time_average_stats`

The C **math library** must be linked if math operations are programmed. This might require setting the `'-lm'` option in the compilation statement, e.g.,

```
cc    -lm    myprogram.c
```

If you are using the Law & Kelton 0-1 random number generator be sure to reference the header file `'lcgrand.h'` (and put the `lcgrand.c` and `lcgrand.h` files in your working directory):

```
#include "lcgrand.h"
```

If you do not use dynamic storage allocation, be sure to allocated sufficient storage space for queues and test for queue overflow.

Use **file pointers** to allow files to be opened/closed from within code rather than at the operating system level, e.g.,

```
FILE *infile, *outfile;  
infile  = fopen ( "input.dat", "r" );  
outfile = fopen ( "output.dat", "w" );  
fclose ( infile );
```

Remember, in C, array indices start at zero. For code clarity, you may wish to skip the 0-th array position, so that for example indices can match up with event numbers.

Example: We consider Law and Kelton's single-server queuing system C-code. It follows essentially the material of Lecture 2. Any differences will be pointed out.

External definitions: In C, external definitions allocated storage for variables and arrays that can be accessed by all subprograms. Declarations of these, therefor, need to be made only once. This is not the case for other languages.

```
/* External definitions for single-server queuing 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);
```

Note the following:

1. Standard I/O and math library headers need to be invoked.
2. BUSY and IDLE mnemonics.
3. Descriptive variable/array names.
4. FILE definitions.
5. Non-returning subproblems and returning functions.

Main function (program): This is the body of the simulation program. It includes the following important steps: Setting of the number of different events, reading of input parameters from file, writing the report heading, initializing the simulation, implementing the main simulation loop and generating the final report.

```
main() /* Main function. */
{
    /* Open input and output files. */
    infile = fopen("mml.in", "r");
    outfile = fopen("mml.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 queuing system\n\n");
    fprintf(outfile, "Mean inter-arrival 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;
}
```

Note the following:

1. Instead of reading the input parameters, one could query the console and enter these manually.
2. Use of file pointers to open files within the program-.
3. Initialization of the number of event types would be better done by the following: '#define num_events 2'
4. The '&'s in fscanf are important since these are pointers to the appropriate locations in storage.
5. '\n' is the newline instruction. Unlike some other languages, a newline is **not** automatic when a new print or read instruction is given.
6. There is an extensive use of /* ... */ for notational comments.
7. The while loop is structured for clarity.
8. 'break' is necessary to terminate actions in the switch statement. 'break' is a way of departing from a 'while', 'do while', 'for' or a 'switch' statement.
9. It is good practice to 'close' files before exiting.

Initialization function (subprogram): Here, all registers are initialized. (See computer representation of the hand simulation of the last lecture.)

```
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;
}
```

Note the following:

1. 'num_custs_delayed' would better be 'num_custs_entered_svc'.
2. 'time_next_event' doesn't use the 0th register.
3. Random time could be a set of fixed values just like the example of the previous lecture.
4. Note especially the ingenious use of a very large number (1.0e+30) as a marker for the situation that 'time_next_event[2]' is empty, i.e., there is no next event of type 2 (type2 = departure).

Timing function (subprogram): This is used to find the trigger event defined earlier.

```
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 simul. Clock.*/
    sim_time = min_time_next_event;
}
```

Note the following:

- a) Local variables must be declared.
- b) In any determination of a minimum, a dummy very large value must be used to start the comparison process. This value is $1.0e+29 < 1.0e+30$.

- c) List processing could be used here. But, since there are only a few (2) alternatives, a simple test will do.
- d) Note the fact that carriage return has no effect on the meaning to the compiler. The compiler looks for '{...}', /*...*/ and ';'.
- e) A test for empty event list is added for safety. Recall that a value of $1.0e+30$ means **no** entry.

Arrival event function: This function follows the flowchart of page 12 of the last lecture notes, with the exception that 'gather statistics' has been moved to a separate function earlier in the main loop.

```
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 the 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);
    }
}
```

Note the following:

- a) Here, too, a randomized service time calculation could be replaced by an input value.
- b) The operator ++ adds 1 to the integer contained in 'num_in_q'.
- c) The '+=' operator acts to add the contents of 'delay' to the contents of 'total_of_delays'.
- d) The use of mnemonics helps the readability.

Depart event function: The logic for this was shown in the flowchart on page 13 of the previous lecture notes.

```
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 no. 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. */

        time_arrival[i] = time_arrival[i + 1];
    }
}
```

Note the following:

- a) If at the point that the queue is empty, the instruction
`time_next_event[2] = 1.0e+30;`
were omitted, the program would go into an infinite loop. The reason is that, without this marking, the 'event' (actually, its time) would remain on the event list. The timing function would return it as the 'next' triggering event and the loop would never end.
- b) The 'for' loop (`for (i = 1; i <= num_in_q; ++i);`) moves each customer up one place (toward the server). This assures that the arrival time of the customer to next enter service is always found in 'time_arrival[1]'. Managing the queue this way is inefficient and could be improved with the use of pointers.
- c) The subtraction of 'time_arrival[1]' from the clock value 'sim_time' to obtain the delay in queue can create a problem if the simulation is run a long time. These two quantities were declared as 'float' with perhaps 8 places of decimal accuracy (Sun). There is potential for serious loss of precision. If we declare both these quantities as 'double', we likely can avoid such a problem.

Report generator function:

```
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);
}
```

Nothing to explain here.

Update_time_avg_stats function: This updates the areas under the two functions $Q(t)$ and $B(t)$ needed for continuous time statistics.

```
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;
}
```

Note the following:

- a) This function is invoked before processing either type of event.
- b) If you recall, there were two other statistics that required updating, 'num_custs_delayed' (i.e., number of customers that entered service) and 'total_of_delays'. These are not included in this function. Due to the logic required to update them, they are better handled in the event routines.

The Law & Kelton C-code just described will not run without the function 'lcgrand.c', i.e., the random number generator. This function, as well as the Law & Kelton C-code (called 'mm1.c') and the input file (mm1.in) is provided on the web site:

<http://www.mhhe.com/lawkelton>.

On a UNIX system, mm1.c and lcgrand.c may be compiled and linked with the following instruction:

```
cc -lm mm1.c lcgrand.c
```

This produces an executable file 'a.out' resulting in the following output:

Several other things become evident when perusing the code:

- a) The inputs to the simulation (i.e., the inter-arrival and service times) are entered in the code itself rather than generated by a random number generator.
- b) A header file 'errno.h' for out-of-range error checking is introduced.
- c) Array sizes are enlarged and variables are declared for a two-node system to be simulated in the homework.
- d) The code itself is not annotated to any extent and is much harder to follow.
- e) Most of the variable and array names are not self-explanatory, nor are there mnemonics that would make it easier to follow what is going on.

Nevertheless, the system being simulated is identical to the one in the Law & Kelton example. One way to check this is to enter the following data into the initial (leftmost) entries in the 'ia' and 's1' float arrays:

```
float ia[40]={0.4,1.2,0.5,1.7,0.2,1.6,0.2,1.4,1.9,...
float s1[30]={2.0,0.7,0.2,1.1,3.7,0.6,...
```

and change the number of customers handled from 15 to 5. The results are identical to the Law & Kelton results:

t	e'	x1	gone	Event List

0.00	INIT	0	0	(a, 0.40)
0.40	a	1	0	(a, 1.60) (d1, 2.40)
1.60	a	2	0	(a, 2.10) (d1, 2.40)
2.10	a	3	0	(a, 3.80) (d1, 2.40)
2.40	d1	2	1	(a, 3.80) (d1, 3.10)
3.10	d1	1	2	(a, 3.80) (d1, 3.30)
3.30	d1	0	3	(a, 3.80)
3.80	a	1	3	(a, 4.00) (d1, 4.90)
4.00	a	2	3	(a, 5.60) (d1, 4.90)
4.90	d1	1	4	(a, 5.60) (d1, 8.60)
5.60	a	2	4	(a, 5.80) (d1, 8.60)
5.80	a	3	4	(a, 7.20) (d1, 8.60)
7.20	a	4	4	(a, 9.10) (d1, 8.60)
8.60	d1	3	5	(a, 9.10) (d1, 9.20)

AST = 2.080				
TP = 0.581				

with the exception that the statistics gathered are different. Here, AST means Average time in system and TP means Throughput.

```

main()
{
    initialization();
    while ( gone < 15 )
    {
        triggering_event();
        t = et[e];
        if(e==0) new_arrival();
        else departure1();
        printf("\n%6.2f ",t);
        if(e==0) printf("  a  ");
        else printf("  d1 ");
        printf(" %4d %3d ",x1,gone);
        printf(" (a,%5.2f)",et[0]);
        if(x1>0) printf(" (d1,%5.2f)",et[1]);
    }
    ast=0;
    for(i=0;i<15;i++) ast+=(de_t[i]-ar_t[i]);
    ast/=15;
    printf("\n-----");
    printf("\n AST = %6.3f",ast);
    printf("\n TP = %6.3f\n",15/t);
}

```

At the end of the main loop note the statement:

```
for(i=0;i<15;i++) ast+=(de_t[i]-ar_t[i]);
```

This is a loop that is used to add up the total time that customers spend in the system. The important point here is that we have stored the departure and arrival times of each customer in arrays 'de_t' and 'ar_t'. Clearly, this is a poor way to keep statistics. It was just done for expediency in this example. It would be absurd to gather statistics in this way if the simulation run were long or if the simulation were complex.

```

void initialization()
{
    t=0.0;
    x1=0;
    cta=ct1=0;
    arrive=gone=0;
    et[0]=t+ia[cta];
    cta++;
    et[1]=0;
    printf("\n  t      e'      x1  gone      Event List");
    printf("\n-----");
    printf("\n%6.2f ",t);
    printf(" INIT ");
    printf(" %4d %3d ",x1,gone);
    printf(" (a,%5.2f)",et[0]);
}

```

Note that customer numbers start with 0 (perhaps not a good idea). The variable cta holds the number of the next customer to arrive. So, after we determine the arrival time for this customer, we increment this variable to get ready for the next customer.

```

void triggering_event()
{
    e=0;
    min_et = et[0];
    if(x1>0 && et[1]<min_et) {e = 1; min_et = et[1];}
}

```

In the triggering event routine, we find which event in the event list is imminent (0 or 1). We also take note of the time of this imminent (triggering) event.

```

void new_arrival()
{
    x1 = x1+1;
    et[0]=t+ia[cta];
    cta++;
    if(x1==1)
    {
        et[1]=t+s1[ct1];
        ct1++;
    }
    ar_t[arrive]=t;
    arrive++;
}

```

```

void departure1()
{
    x1 = x1-1;
    if(x1>0)
    {
        et[1]=t+s1[ct1];
        ct1++;
    }
    de_t[gone] = t;
    gone = gone+1;
}

```

Note that there is a big difference between this and the Law & Kelton way of handling the arrival time list. In Law & Kelton the list contents are shifted downward one index with each departure, so that the list can never grow larger than the maximum queue length.

In this example, the list grows with each additional arrival and the location of the next arrival time is done with a pointer 'cta'. This implies a certain degree of efficiency; as we pointed out earlier that shifting the list contents is inefficient. Yes, the pointer is efficient. But, the unconstrained growth of the arrival list is equally undesirable, except for short simulation runs.

Other than the arrival list handling issue, the coding for the arrival and departure events in both simulation codes is not all that different.

Homework Assignment #3 due 3:00 PM, Thursday, September 28, 1999

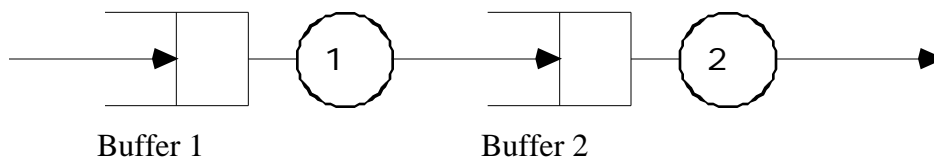
Problem 1.

Download the C-code for one-node system from our class web page. Compare this C-code to that (of Law & Kelton) for the single-server queue system covered in this lecture (Lecture 3).

- Annotate the one-node system C-code in a manner consistent with the detail of the Law & Kelton C-code.
- Modify the names of variables and arrays to make them descriptive and add mnemonics where applicable. Also, enhance printout to better describe the output statistics.

Problem 2.

Consider the following two-node system:



There are three types of events: arrival event (a), departure at node 1 (d_1), and departure at node 2 (d_2). The sizes of the waiting buffer space at both nodes are infinite. Suppose the inter-arrival times are 1.0 for all customers (So, the arrival times of customers are 1.0, 2.0, 3.0, 4.0, ...). The service times at node 1 are: 2.1, 2.0, 2.0, 2.0, 0.51, 0.51, 0.51, 0.51, 0.51, 0.51, 0.51, 0.51, 0.51, 0.51, ... The service times at node 2 are: 1.0, 2.1, 3.0, 3.0, 3.5, 3.5, 0.1, 0.2, 0.3, 0.43, 0.43, 0.43, 0.43, 0.43, 0.43, 0.43, ...

Assumption: For simplicity, if any two or three events have the same event times, we assume that event (a) happens before (d_1), and (d_1) happens before (d_2). This implies (a) happens before (d_2).

Modify the one-node system C-code (from class web page) to simulate this system. Add the # of customers at node 2 to the printout (see result below). Stop this simulation when the 15th customer leaves this system and calculate the average time in system and throughput (average number of customers served per unit time, i.e., $\frac{15}{\text{Total Simulation Time}}$).

You must hand in a printout of your program, the simulation results, average time in system and throughput. **Note: please keep your programs for future homework problems.**

Below is the result printout of the first 10 stages:

simul. clock	e'	# of customers at			Event List
		node 1	node 2	gone	
0.00	Initialization	0	0	0	(a, 1.00)
1.00	a	1	0	0	(a, 2.00) (d1, 3.10)
2.00	a	2	0	0	(a, 3.00) (d1, 3.10)
3.00	a	3	0	0	(a, 4.00) (d1, 3.10)
3.10	d1	2	1	0	(a, 4.00) (d1, 5.10) (d2, 4.10)
4.00	a	3	1	0	(a, 5.00) (d1, 5.10) (d2, 4.10)
4.10	d2	3	0	1	(a, 5.00) (d1, 5.10)
5.00	a	4	0	1	(a, 6.00) (d1, 5.10)
5.10	d1	3	1	1	(a, 6.00) (d1, 7.10) (d2, 7.20)
6.00	a	4	1	1	(a, 7.00) (d1, 7.10) (d2, 7.20)
7.00	a	5	1	1	(a, 8.00) (d1, 7.10) (d2, 7.20)
etc.					

Hints:

1. The following pseudo-C-code will be very helpful.
2. Finish the simulation by hand before doing the computer program.

Pseudo-C-code for this problem

```
#include <math.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>

float et[3];
float t;
float min_et, ast;
int x1, x2;
int arrive, gone;
int e;
int i;
float ia[40]={ 1, 1, 1, 1, ..... };
float s1[30]={ ... };
float s2[25]={ ... };
int cta, ct1, ct2;
float ar_t[40];
float de_t[15];
```

```

main{
INITIALIZATION
while ( gone < 15 )
{

$$e' = \arg \min_{e_i \in (Q,B)} \{t_i\};$$

t = et[e];
if( e==0) new_arrival();
else if(e==1) departure1();
else departure2();
Print out the simulation status in this iteration
}
Calculate Ave. ST and TP
}

*  $e' = \arg \min_{e_i \in (Q,B)} \{t_i\}$  *
e=0;
min_et = et[0];
if(x1 > 0 && et[1] < min_et) {e = 1; min_et = et[1];}
if(x2 > 0 && et[2] < min_et) {e = 2; min_et = et[2];}

* new_arrival() *
x1 = x1+1;
update et[0];
cta = cta+1;
if(x1==1) {update et[1]; ct1 = ct1+1;}
update the arrival time statistics

* departure1() *
x1 = x1-1;
x2 = x2+1;
if(x1>=1) {update et[1]; ct1 = ct1+1;}
if(x2==1) {update et[2]; ct2 = ct2+1;}

* departure2() *
x2 = x2-1;
gone = gone+1;
if(x2>=1) {update et[2]; ct2 = ct2+1;}
update the departure time statistics

* update et[2] *
et[2] = t + s2[ct2];

```