

EECE 5643: Simulation and Performance Evaluation
Professor Ningfang Mi

Homework 3

- Assignment Due: 02/16/2023 -

Harrison Sun
Monday, Thursday 11:45 am - 1:25 pm
Completed: February 16, 2023

1 Ex. 3.1.1

- Modify program `ssq2` to use *Exponential*(1.5) service times.
- Process a relatively large number of jobs, say 100000, and report what changes this produces relative to the statistics in Example 3.1.3.
- Explain (or conjecture) why some statistics change and others do not.

	123456	123456789	975312468	97531	246810
interarrival time	2.00	2.00	2.00	2.00	2.00
wait time	5.99	6.04	6.10	5.95	6.01
delay time	4.49	4.53	4.60	4.45	4.51
service time	1.50	1.50	1.50	1.50	1.51
number in node	3.00	3.02	3.05	2.97	3.00
number in queue	2.25	2.27	2.30	2.22	2.25
utilization	0.75	0.75	0.75	0.75	0.75

Discrete-Event Simulation Example 3.1.3

- The theoretical averages for a single-server service node using *Exponential*(2.0) arrivals and *Uniform*(1.0, 2.0) service times are

\bar{r}	\bar{w}	\bar{d}	\bar{s}	\bar{l}	\bar{q}	\bar{x}
2.00	3.83	2.33	1.50	1.92	1.17	0.75

- Although the server is busy 75% of the time, on average there are approximately two jobs in the service node
- A job can expect to spend more time in the queue than in service
- To achieve these averages, many jobs must pass through node

Navigation icons: back, forward, search, etc.

The average interarrival time (\bar{r}), average service time (\bar{s}), and utilization (\bar{x}) remain the same. The average wait time (\bar{w}), average delay time (\bar{d}), average number in the node (\bar{l}), and average number in the queue (\bar{q}) increase. This is because the service time distribution changes from a *Uniform*(1.0, 2.0) to an *Exponential*(1.5). While the mean stays the same, the average service time increases due to the significantly higher possible service times caused by the exponential distribution. Whenever this happens, the number in the queue (and thus, the number in the node) increases and propagates to higher wait times as the subsequent jobs arrive.

```

hlsun: Simulation-and-Perforri X + v
(base) hlsun:Simulation-and-Performance-Evaluation$ conda deactivate
hlsun:Simulation-and-Performance-Evaluation$ make clean
rm Homework3.3 Homework3.4 Homework3.1 Homework3.2
hlsun:Simulation-and-Performance-Evaluation$ make
g++ Homework3.3.cpp c_lib/rng.c -o Homework3.3
g++ Homework3.4.cpp c_lib/rvgs.c c_lib/rngs.c -o Homework3.4
g++ Homework3.1.cpp c_lib/rng.c -o Homework3.1
g++ Homework3.2.cpp c_lib/rvgs.c c_lib/rngs.c -o Homework3.2
hlsun:Simulation-and-Performance-Evaluation$ ./Homework3.1 r 100000 s 123456

for 100000 jobs
average interarrival time = 2.00
average wait ..... = 5.99
average delay ..... = 4.49
average service time .... = 1.50
average # in the node ... = 3.00
average # in the queue .. = 2.25
utilization ..... = 0.75
hlsun:Simulation-and-Performance-Evaluation$ ./Homework3.1 r 100000 s 123456789

for 100000 jobs
average interarrival time = 2.00
average wait ..... = 6.04
average delay ..... = 4.53
average service time .... = 1.50
average # in the node ... = 3.02
average # in the queue .. = 2.27
utilization ..... = 0.75
hlsun:Simulation-and-Performance-Evaluation$ ./Homework3.1 r 100000 s 975312468

for 100000 jobs
average interarrival time = 2.00
average wait ..... = 6.10
average delay ..... = 4.60
average service time .... = 1.50
average # in the node ... = 3.05
average # in the queue .. = 2.30
utilization ..... = 0.75
hlsun:Simulation-and-Performance-Evaluation$ ./Homework3.1 r 100000 s 97531

for 100000 jobs
average interarrival time = 2.00
average wait ..... = 5.95
average delay ..... = 4.45
average service time .... = 1.50
average # in the node ... = 2.97
average # in the queue .. = 2.22
utilization ..... = 0.75
hlsun:Simulation-and-Performance-Evaluation$ ./Homework3.1 r 100000 s 246810

for 100000 jobs
average interarrival time = 2.00
average wait ..... = 6.01
average delay ..... = 4.51
average service time .... = 1.51
average # in the node ... = 3.00
average # in the queue .. = 2.25
utilization ..... = 0.75
hlsun:Simulation-and-Performance-Evaluation$ |

```

```

1  /**
2  * Modified by Harrison Sun
3  * sun.har@northeastern.edu
4  * February 11, 2023
5  */
6
7  /*
8  * This program – an extension of program ssq1.c – simulates a single-server
9  * FIFO service node using Exponentially distributed interarrival times and
10 * Uniformly distributed service times (i.e. a M/U/1 queue).
11 *
12 * Name           : ssq2.c (Single Server Queue, version 2)
13 * Author          : Steve Park & Dave Geyer
14 * Language        : ANSI C
15 * Latest Revision : 9–11–98
16 */
17
18
19 #include <exception>
20 #include <iostream>
21 #include <cstdlib>
22 #include <cstring>
23 #include <stdio.h>
24 #include <string>
25 #include <math.h>
26 #include "c-lib/rng.h"
27
28 #define LAST      10000L           /* number of jobs processed */
29 #define START     0.0             /* initial time */
30
31
32 double Exponential(double m)
33 /*
34 * generate an Exponential random variate, use m > 0.0
35 */
36 {
37     return (-m * log(1.0 - Random()));
38 }
39
40
41 double Uniform(double a, double b)
42 /*
43 * generate a Uniform random variate, use a < b
44 */
45 {
46     return (a + (b - a) * Random());
47 }
48
49
50
51 double GetArrival(void)
52 /*
53 * generate the next arrival time
54 */
55 {
56     static double arrival = START;
57
58     arrival += Exponential(2.0);
59     return (arrival);
60 }
61
62
63
64 //double GetService(void)
65

```

```

66 /// *
67 // * generate the next service time
68 // *
69 // */
70 //{
71 //     return (Uniform(1.0, 2.0));
72 //}
73
74 /* Changing the GetService(void) function to return Exponential(1.5) service times */
75 double GetService(void)
76 {
77     /* Generate the next service time */
78
79     return (Exponential(1.5));
80 }
81
82 /* function to check if the input is a number */
83 bool checkArg(char* input)
84 {
85     try
86     {
87         if (strlen(input) > 9)
88         {
89             throw std::logic_error("Number is too large.");
90         }
91
92         for (int i = 0; i < strlen(input); ++i)
93         {
94
95             if (std::isdigit(input[i])) continue;
96             else
97             {
98                 std::string errorMessage;
99                 errorMessage.append((std::string)input);
100                 errorMessage.append(" is not a digit.");
101                 throw std::logic_error(errorMessage);
102             }
103         }
104         return 1;
105     }
106
107     catch (const std::logic_error& error)
108     {
109         std::cerr << error.what() << std::endl;
110         return 0;
111     }
112 }
113
114 int main(int argc, char* argv[])
115 {
116     long    index = 0;
117     double  arrival = START;
118     double  delay;
119     double  service;
120     double  wait;
121     double  departure = START;
122     struct {
123         double delay;
124         double wait;
125         double service;
126         double interarrival;
127     } sum = { 0.0, 0.0, 0.0 };
128
129     long numRuns{};
130
131     /* job index */
132     /* time of arrival */
133     /* delay in queue */
134     /* service time */
135     /* delay + service */
136     /* time of departure */
137     /* sum of ... */
138     /* delay times */
139     /* wait times */
140     /* service times */
141     /* interarrival times */
142
143     /* number of runs */

```

```

131 // Set the seed
132 for (int i = 0; i < argc; ++i)
133 {
134     if (*argv[i] == 's' && checkArg(argv[i + 1]))
135     {
136         PutSeed(std::stol(argv[i + 1]));
137         break;
138     }
139     else
140     {
141         PutSeed(123456789);
142     }
143 }
144
145 // Set the number of runs
146 for (int i = 0; i < argc; ++i)
147 {
148     if (*argv[i] == 'r' && checkArg(argv[i + 1]))
149     {
150         numRuns = std::stol(argv[i + 1]);
151         break;
152     }
153     else
154     {
155         numRuns = 10000;
156     }
157 }
158
159 while (index < numRuns) {
160     index++;
161     arrival = GetArrival();
162     if (arrival < departure)
163         delay = departure - arrival;           /* delay in queue */
164     else
165         delay = 0.0;                           /* no delay */
166     service = GetService();
167     wait = delay + service;
168     departure = arrival + wait;                 /* time of departure */
169     sum.delay += delay;
170     sum.wait += wait;
171     sum.service += service;
172 }
173 sum.interarrival = arrival - START;
174
175 printf("\nfor %ld jobs\n", index);
176 printf("    average interarrival time = %6.2f\n", sum.interarrival / index);
177 printf("    average wait ..... = %6.2f\n", sum.wait / index);
178 printf("    average delay ..... = %6.2f\n", sum.delay / index);
179 printf("    average service time .... = %6.2f\n", sum.service / index);
180 printf("    average # in the node ... = %6.2f\n", sum.wait / departure);
181 printf("    average # in the queue .. = %6.2f\n", sum.delay / departure);
182 printf("    utilization ..... = %6.2f\n", sum.service / departure);
183
184 return (0);
185 }

```

2 Ex. 3.1.5

- (a) Verify that the mean service time in Example 3.1.4 is 1.5.
- (b) Verify that the steady-state statistics in Example 3.1.4 seem to be correct.
- (c) Note that the arrival rate, service rate, and utilization are the same as those in Example 3.1.3. Explain (or conjecture) why this is so. Be Specific.

Discrete-Event Simulation Example 3.1.3

- The theoretical averages for a single-server service node using *Exponential*(2.0) arrivals and *Uniform*(1.0, 2.0) service times are

\bar{r}	\bar{w}	\bar{d}	\bar{s}	\bar{l}	\bar{q}	\bar{x}
2.00	3.83	2.33	1.50	1.92	1.17	0.75

- Although the server is busy 75% of the time, on average there are approximately two jobs in the service node
- A job can expect to spend more time in the queue than in service
- To achieve these averages, many jobs must pass through node

Discrete-Event Simulation Example 3.1.4

- The theoretical steady-state statistics for this model are

\bar{r}	\bar{w}	\bar{d}	\bar{s}	\bar{l}	\bar{q}	\bar{x}
2.00	5.77	4.27	1.50	2.89	2.14	0.75

- The arrival rate, service rate, and utilization are identical to Example 3.1.3
- The other four statistics are significantly larger
- Performance measures are sensitive to the choice of service time distribution

```
hlsun: Simulation-and-Perfor x + v
hlsun:Simulation-and-Performance-Evaluation$ make clean
rm Homework3.3 Homework3.4 Homework3.1 Homework3.2
hlsun:Simulation-and-Performance-Evaluation$ make
g++ Homework3.3.cpp c_lib/rng.c -o Homework3.3
g++ Homework3.4.cpp c_lib/rvgs.c c_lib/rngs.c -o Homework3.4
g++ Homework3.1.cpp c_lib/rng.c -o Homework3.1
g++ Homework3.2.cpp c_lib/rvgs.c c_lib/rngs.c -o Homework3.2
hlsun:Simulation-and-Performance-Evaluation$ ./Homework3.2 r 100000 s 123456

for 100000 jobs
average interarrival time = 2.00
average wait ..... = 5.61
average delay ..... = 4.10
average service time .... = 1.50
average # in the node ... = 2.80
average # in the queue .. = 2.05
utilization ..... = 0.75
hlsun:Simulation-and-Performance-Evaluation$ ./Homework3.2 r 100000 s 246810

for 100000 jobs
average interarrival time = 2.00
average wait ..... = 5.82
average delay ..... = 4.31
average service time .... = 1.50
average # in the node ... = 2.90
average # in the queue .. = 2.15
utilization ..... = 0.75
hlsun:Simulation-and-Performance-Evaluation$ ./Homework3.2 r 100000 s 1357911

for 100000 jobs
average interarrival time = 2.00
average wait ..... = 5.58
average delay ..... = 4.09
average service time .... = 1.50
average # in the node ... = 2.79
average # in the queue .. = 2.04
utilization ..... = 0.75
hlsun:Simulation-and-Performance-Evaluation$ ./Homework3.2 r 100000 s 987654321

for 100000 jobs
average interarrival time = 2.00
average wait ..... = 5.73
average delay ..... = 4.23
average service time .... = 1.50
average # in the node ... = 2.86
average # in the queue .. = 2.11
utilization ..... = 0.75
hlsun:Simulation-and-Performance-Evaluation$ ./Homework3.2 r 100000 s 12357

for 100000 jobs
average interarrival time = 2.00
average wait ..... = 5.66
average delay ..... = 4.16
average service time .... = 1.49
average # in the node ... = 2.82
average # in the queue .. = 2.08
utilization ..... = 0.75
hlsun:Simulation-and-Performance-Evaluation$ |
```


	123456	246810	1357911	987654321	12357
interarrival time	2.00	2.00	2.00	2.00	2.00
wait time	5.61	5.82	5.58	5.73	5.66
delay time	4.10	4.31	4.09	4.23	4.16
service time	1.50	1.50	1.50	1.50	1.49
number in node	2.80	2.90	2.79	2.86	2.82
number in queue	2.05	2.15	2.04	2.11	2.08
utilization	0.75	0.75	0.75	0.75	0.75

The arrival rate is the same because the interarrival time is defined by the same distribution as in Example 3.1.3: $r \sim \text{Exponential}(2.0)$.

The average service rate is the same as in Example 3.1.3 due to the distribution of the service times. The average number of tasks \bar{t} of $t \sim 1 + \text{Geometric}(0.9)$ is 1 plus the inverse of $p = 0.9$. Therefore, the average number of tasks is 10. The average of $\text{Uniform}(0.1, 0.2)$ is 0.15. Multiplying this and the number of tasks together results in an average service rate $\bar{s} = 1.5$, which is the same as the service rate in Example 3.1.3.

The server utilization is the same as in Example 3.1.3 because the average interarrival times and average service rates are the same in both examples. The server utilization is a ratio of the interarrival and service time averages.

```

1  /**
2  * Homework 3.2
3  * EECE 5643 – Simulation and Performance Evaluation
4  * Author: Harrison Sun
5  * Email: sun.har@northeastern.edu
6  */
7
8  #include <cstdlib>
9  #include <cstring>
10 #include <stdio.h>
11 #include <exception>
12 #include <iostream>
13 #include <math.h>
14 #include <string>
15 #include "c_lib/rvgs.h"
16 #include "c_lib/rngs.h"
17
18 #define LAST          10000L          /* number of jobs processed */
19 #define START         0.0             /* initial time             */
20
21 /**
22 * double GetArrival()
23 *
24 * @param void
25 * @return arrival – the next arrival time
26 *
27 * This function calculates the arrival times for each process.
28 */
29
30 double GetArrival()
31 {
32     static double arrival = START;
33
34     arrival += Exponential(2.0);
35     return (arrival);
36 }
37
38
39 /**
40 * double GetService()
41 *
42 * @param void
43 * @return sum – the total service time for the process
44 *
45 * This function calculates the service times for each process.
46 */
47
48 double GetService()
49 {
50     long k{};
51     double sum{ 0.0 };
52     long tasks = 1 + Geometric(0.9);
53     for (k = 0; k < tasks; ++k)
54     {
55         sum += Uniform(0.1, 0.2);
56     }
57     return sum;
58 }
59
60 /**
61 * bool checkArg()
62 *
63 * @param char* input – the input string literal from the console
64 * @return bool – true if the input is a number, false otherwise
65 */

```

```

66  * This function determines whether the argument is a number.
67  */
68
69  bool checkArg(char* input)
70  {
71      try
72      {
73          if (strlen(input) > 9)
74          {
75              throw std::logic_error("Number is too large.");
76          }
77
78          for (int i = 0; i < strlen(input); ++i)
79          {
80
81              if (std::isdigit(input[i])) continue;
82              else
83              {
84                  std::string errorMessage;
85                  errorMessage.append((std::string)input);
86                  errorMessage.append(" is not a digit.");
87                  throw std::logic_error(errorMessage);
88              }
89          }
90          return 1;
91      }
92
93      catch (const std::logic_error& error)
94      {
95          std::cerr << error.what() << std::endl;
96          return 0;
97      }
98  }
99
100 /**
101  * int main()
102  *
103  * @param int argc - the number of arguments
104  * @param char* argv[] - the arguments
105  *
106  * @return int - 0 if the program runs successfully
107  */
108
109 int main(int argc, char* argv[])
110 {
111     long index = 0;
112     double arrival = START;
113     double delay;
114     double service;
115     double wait;
116     double departure = START;
117     struct {
118         double delay;
119         double wait;
120         double service;
121         double interarrival;
122     } sum = { 0.0, 0.0, 0.0 };
123
124     long numRuns{};
125
126     // Set the seed
127     for (int i = 0; i < argc; ++i)
128     {
129         if (*argv[i] == 's' && checkArg(argv[i + 1]))
130         {

```

```

131         PutSeed(std::stoi(argv[i + 1]));
132         break;
133     }
134     else
135     {
136         PutSeed(123456789);
137     }
138 }
139
140 // Set the number of runs
141 for (int i = 0; i < argc; ++i)
142 {
143     if (*argv[i] == 'r' && checkArg(argv[i + 1]))
144     {
145         numRuns = std::stoi(argv[i + 1]);
146         break;
147     }
148     else
149     {
150         numRuns = 10000;
151     }
152 }
153
154 while (index < numRuns) {
155     index++;
156     arrival = GetArrival();
157     if (arrival < departure)
158         delay = departure - arrival;           /* delay in queue */
159     else
160         delay = 0.0;                           /* no delay */
161     service = GetService();
162     wait = delay + service;
163     departure = arrival + wait;                 /* time of departure */
164     sum.delay += delay;
165     sum.wait += wait;
166     sum.service += service;
167 }
168 sum.interarrival = arrival - START;
169
170 printf("\nfor %ld jobs\n", index);
171 printf("    average interarrival time = %6.2f\n", sum.interarrival / index);
172 printf("    average wait ..... = %6.2f\n", sum.wait / index);
173 printf("    average delay ..... = %6.2f\n", sum.delay / index);
174 printf("    average service time .... = %6.2f\n", sum.service / index);
175 printf("    average # in the node ... = %6.2f\n", sum.wait / departure);
176 printf("    average # in the queue .. = %6.2f\n", sum.delay / departure);
177 printf("    utilization ..... = %6.2f\n", sum.service / departure);
178 return 0;
179 }

```

3 Ex. 3.3.1

Let β be the probability of feedback and let the integer-valued random variable \mathcal{X} be the number of times a job feeds back.

(a) For $x = 0, 1, 2, \dots$ what is $Pr(\mathcal{X} = x)$?

The probability of feedback $Pr(\mathcal{X} = x)$ is equal to $\beta^x \times (1 - \beta)$. That is, the probability of each feedback is defined by β and the single successful job completion is defined by $1 - \beta$.

(b) How does this relate to the discussion of acceptance/rejection in Section 2.3 (i.e., Example 2.3.8)?

The probability of feedback is analogous to rejection of the job. Therefore, it can be modeled similarly to the acceptance/rejection model with the acceptance criteria encompassing $1 - \beta$ of the feature space and β encompassing the remainder (outside of the acceptance area) of the feature space. In this case, a two dimensional feature space is not required and can be flattened to a single line.

```

1  /**
2  * Homework 3.3
3  * EECE 5643 – Simulation and Performance Evaluation
4  * Author: Harrison Sun
5  * Email: sun.har@northeastern.edu
6  */
7
8  #define DEFAULTBETA 0.9          // default value for beta
9  #define DEFAULTRUNS 100000L     // default value for number of runs
10
11 #include <cstdlib>
12 #include <cstring>
13 #include <stdio.h>
14 #include <exception>
15 #include <iostream>
16 #include <math.h>
17 #include <string>
18 #include <vector>
19 #include "c_lib/rng.h"
20
21 /**
22 * bool checkArg()
23 *
24 * @param char* input – the input string literal from the console
25 * @return bool – true if the input is a number, false otherwise
26 *
27 * This function determines whether the argument is a number.
28 */
29
30 bool checkArg(char* input)
31 {
32     try
33     {
34         if (strlen(input) > 9)
35         {
36             throw std::logic_error("Number is too large.");
37         }
38
39         for (int i = 0; i < strlen(input); ++i)
40         {
41
42             if (std::isdigit(input[i]) || (input[i] == '.')) continue;
43             else
44             {
45                 std::string errorMessage;
46                 errorMessage.append((std::string)input);
47                 errorMessage.append(" is not a number.");
48                 throw std::logic_error(errorMessage);
49             }
50         }
51         return 1;
52     }
53
54     catch (const std::logic_error& error)
55     {
56         std::cerr << error.what() << std::endl;
57         return 0;
58     }
59 }
60
61 /**
62 * int main()
63 *
64 * @param int argc – the number of arguments
65 * @param char* argv[] – the arguments

```

```

66  *
67  * @return int -- returns 0 if the program runs successfully
68  */
69
70 int main(int argc, char* argv[])
71 {
72     long numRuns{};
73     long Beta{};
74     std::vector<int> feedback;
75
76     // Set the seed
77     for (int i = 0; i < argc; ++i)
78     {
79         if (*argv[i] == 's' && checkArg(argv[i + 1]))
80         {
81             PutSeed(std::stol(argv[i + 1]));
82             break;
83         }
84         else
85         {
86             PutSeed(123456789);
87         }
88     }
89
90     // Set the number of runs
91     for (int i = 0; i < argc; ++i)
92     {
93         if (*argv[i] == 'r' && checkArg(argv[i + 1]))
94         {
95             numRuns = std::stol(argv[i + 1]);
96             break;
97         }
98         else
99         {
100             numRuns = DEFAULT_RUNS;
101         }
102     }
103
104     // Set the number of runs
105     for (int i = 0; i < argc; ++i)
106     {
107         if (*argv[i] == 'B' && checkArg(argv[i + 1]))
108         {
109             Beta = std::stol(argv[i + 1]);
110             break;
111         }
112         else
113         {
114             Beta = DEFAULT_BETA;
115         }
116     }
117
118     for (int i = 0; i < numRuns; ++i)
119     {
120         int numFeedback{ 0 };
121         double random = Random();
122         /* Test if feedback */
123         while (random > Beta)
124         {
125             numFeedback++;
126             random = Random();
127         }
128         if (numFeedback < feedback.size())
129             feedback[numFeedback]++;
130         else

```

```

131     {
132         feedback.resize(numFeedback + 1);
133         feedback[numFeedback]++;
134     }
135 }
136 std::cout << feedback.size() << std::endl;
137 for (int i = 0; i < feedback.size(); ++i)
138 {
139     feedback[i] /= numRuns;
140     std::cout << "Probability of " << i << " feedbacks: " << feedback[i] << std::endl;
141 }
142
143 return 0;
144 }

```


4 Ex. 3.3.4

Modify program ssq2 to account for a finite queue capacity.

(a) For the queue capacities 1,2,3,4,5, and 6, construct a table of the estimated steady-state probability of rejection.

Uniform(1.0, 2.0)	123456	246810	97531
Rejection Q = 1	18.53	18.45	18.19
Rejection Q = 2	8.98	9.05	8.78
Rejection Q = 3	4.77	4.76	4.55
Rejection Q = 4	2.65	2.61	2.51
Rejection Q = 5	1.45	1.52	1.35
Rejection Q = 6	0.83	0.85	0.74

(b) Also, construct a similar table if the service-time distribution is changed to be *Uniform*(1.0, 3.0).

Uniform(1.0, 3.0)	123456	246810	97531
Rejection Q = 1	27.66	27.58	27.24
Rejection Q = 2	18.45	18.30	18.11
Rejection Q = 3	13.82	13.69	13.41
Rejection Q = 4	10.93	10.77	10.55
Rejection Q = 5	9.05	8.86	8.60
Rejection Q = 6	7.74	7.55	7.43

(c) Comment on how the probability of rejection depends on the service process.

The probability of rejection increases when the service time increases. This makes sense, as a higher service time allows the queue to build up, particularly when the interarrival rate is faster than the service time.

(d) How did you convince yourself these tables are correct?

This makes sense, as the increased service time increases the rejection rate. Additionally, a longer queue makes the rejection rate decrease, as the service node can handle multiple jobs arriving in a short period of time given that the queue has space.

```
hlsun: Simulation-and-Perfor x + v
hlsun:Simulation-and-Performance-Evaluation$ ./Homework3.4 r 100000 s 123456 q 1 u 1.0 2.0
percent of rejections... = 18.53
utilization ..... = 0.61
hlsun:Simulation-and-Performance-Evaluation$ ./Homework3.4 r 100000 s 123456 q 2 u 1.0 2.0
percent of rejections... = 8.98
utilization ..... = 0.68
hlsun:Simulation-and-Performance-Evaluation$ ./Homework3.4 r 100000 s 123456 q 3 u 1.0 2.0
percent of rejections... = 4.77
utilization ..... = 0.72
hlsun:Simulation-and-Performance-Evaluation$ ./Homework3.4 r 100000 s 123456 q 4 u 1.0 2.0
percent of rejections... = 2.65
utilization ..... = 0.73
hlsun:Simulation-and-Performance-Evaluation$ ./Homework3.4 r 100000 s 123456 q 5 u 1.0 2.0
percent of rejections... = 1.45
utilization ..... = 0.74
hlsun:Simulation-and-Performance-Evaluation$ ./Homework3.4 r 100000 s 123456 q 6 u 1.0 2.0
percent of rejections... = 0.83
utilization ..... = 0.75
hlsun:Simulation-and-Performance-Evaluation$ ./Homework3.4 r 100000 s 246810 q 1 u 1.0 2.0
percent of rejections... = 18.45
utilization ..... = 0.61
hlsun:Simulation-and-Performance-Evaluation$ ./Homework3.4 r 100000 s 246810 q 2 u 1.0 2.0
percent of rejections... = 9.05
utilization ..... = 0.68
hlsun:Simulation-and-Performance-Evaluation$ ./Homework3.4 r 100000 s 246810 q 3 u 1.0 2.0
percent of rejections... = 4.76
utilization ..... = 0.71
hlsun:Simulation-and-Performance-Evaluation$ ./Homework3.4 r 100000 s 246810 q 4 u 1.0 2.0
percent of rejections... = 2.61
utilization ..... = 0.73
hlsun:Simulation-and-Performance-Evaluation$ ./Homework3.4 r 100000 s 246810 q 5 u 1.0 2.0
percent of rejections... = 1.52
utilization ..... = 0.74
hlsun:Simulation-and-Performance-Evaluation$ ./Homework3.4 r 100000 s 246810 q 6 u 1.0 2.0
percent of rejections... = 0.85
utilization ..... = 0.74
hlsun:Simulation-and-Performance-Evaluation$ ./Homework3.4 r 100000 s 97531 q 1 u 1.0 2.0
percent of rejections... = 18.19
utilization ..... = 0.61
hlsun:Simulation-and-Performance-Evaluation$ ./Homework3.4 r 100000 s 97531 q 2 u 1.0 2.0
percent of rejections... = 8.78
utilization ..... = 0.68
hlsun:Simulation-and-Performance-Evaluation$ ./Homework3.4 r 100000 s 97531 q 3 u 1.0 2.0
percent of rejections... = 4.55
utilization ..... = 0.71
hlsun:Simulation-and-Performance-Evaluation$ ./Homework3.4 r 100000 s 97531 q 4 u 1.0 2.0
percent of rejections... = 2.51
utilization ..... = 0.73
hlsun:Simulation-and-Performance-Evaluation$ ./Homework3.4 r 100000 s 97531 q 5 u 1.0 2.0
percent of rejections... = 1.35
utilization ..... = 0.74
hlsun:Simulation-and-Performance-Evaluation$ ./Homework3.4 r 100000 s 97531 q 6 u 1.0 2.0
percent of rejections... = 0.74
utilization ..... = 0.74
hlsun:Simulation-and-Performance-Evaluation$ |
```

```
hlsun: Simulation-and-Perfor x + v
hlsun:Simulation-and-Performance-Evaluation$ ./Homework3.4 r 100000 s 123456 q 1 u 1.0 3.0
percent of rejections... = 27.66
utilization ..... = 0.73
hlsun:Simulation-and-Performance-Evaluation$ ./Homework3.4 r 100000 s 123456 q 2 u 1.0 3.0
percent of rejections... = 18.45
utilization ..... = 0.82
hlsun:Simulation-and-Performance-Evaluation$ ./Homework3.4 r 100000 s 123456 q 3 u 1.0 3.0
percent of rejections... = 13.82
utilization ..... = 0.86
hlsun:Simulation-and-Performance-Evaluation$ ./Homework3.4 r 100000 s 123456 q 4 u 1.0 3.0
percent of rejections... = 10.93
utilization ..... = 0.89
hlsun:Simulation-and-Performance-Evaluation$ ./Homework3.4 r 100000 s 123456 q 5 u 1.0 3.0
percent of rejections... = 9.05
utilization ..... = 0.91
hlsun:Simulation-and-Performance-Evaluation$ ./Homework3.4 r 100000 s 123456 q 6 u 1.0 3.0
percent of rejections... = 7.74
utilization ..... = 0.92
hlsun:Simulation-and-Performance-Evaluation$ ./Homework3.4 r 100000 s 246810 q 1 u 1.0 3.0
percent of rejections... = 27.58
utilization ..... = 0.72
hlsun:Simulation-and-Performance-Evaluation$ ./Homework3.4 r 100000 s 246810 q 2 u 1.0 3.0
percent of rejections... = 18.30
utilization ..... = 0.81
hlsun:Simulation-and-Performance-Evaluation$ ./Homework3.4 r 100000 s 246810 q 3 u 1.0 3.0
percent of rejections... = 13.69
utilization ..... = 0.86
hlsun:Simulation-and-Performance-Evaluation$ ./Homework3.4 r 100000 s 246810 q 4 u 1.0 3.0
percent of rejections... = 10.77
utilization ..... = 0.89
hlsun:Simulation-and-Performance-Evaluation$ ./Homework3.4 r 100000 s 246810 q 5 u 1.0 3.0
percent of rejections... = 8.86
utilization ..... = 0.91
hlsun:Simulation-and-Performance-Evaluation$ ./Homework3.4 r 100000 s 246810 q 6 u 1.0 3.0
percent of rejections... = 7.55
utilization ..... = 0.92
hlsun:Simulation-and-Performance-Evaluation$ ./Homework3.4 r 100000 s 97531 q 1 u 1.0 3.0
percent of rejections... = 27.24
utilization ..... = 0.72
hlsun:Simulation-and-Performance-Evaluation$ ./Homework3.4 r 100000 s 97531 q 2 u 1.0 3.0
percent of rejections... = 18.11
utilization ..... = 0.81
hlsun:Simulation-and-Performance-Evaluation$ ./Homework3.4 r 100000 s 97531 q 3 u 1.0 3.0
percent of rejections... = 13.41
utilization ..... = 0.86
hlsun:Simulation-and-Performance-Evaluation$ ./Homework3.4 r 100000 s 97531 q 4 u 1.0 3.0
percent of rejections... = 10.55
utilization ..... = 0.89
hlsun:Simulation-and-Performance-Evaluation$ ./Homework3.4 r 100000 s 97531 q 5 u 1.0 3.0
percent of rejections... = 8.60
utilization ..... = 0.91
hlsun:Simulation-and-Performance-Evaluation$ ./Homework3.4 r 100000 s 97531 q 6 u 1.0 3.0
percent of rejections... = 7.43
utilization ..... = 0.92
hlsun:Simulation-and-Performance-Evaluation$ |
```

```

1  /**
2  * Modified by Harrison Sun
3  * sun.har@northeastern.edu
4  * February 11, 2023
5  */
6
7  /*
8  * This program is a next-event simulation of a single-server FIFO service
9  * node using Exponentially distributed interarrival times and Uniformly
10 * distributed service times (i.e., a M/U/1 queue). The service node is
11 * assumed to be initially idle, no arrivals are permitted after the
12 * terminal time STOP, and the service node is then purged by processing any
13 * remaining jobs in the service node.
14 *
15 * Name          : ssq3.c (Single Server Queue, version 3)
16 * Author         : Steve Park & Dave Geyer
17 * Language       : ANSI C
18 * Latest Revision : 10-19-98
19 *
20 */
21
22 #include <cstdlib>
23 #include <cstring>
24 #include <stdio.h>
25 #include <exception>
26 #include <iostream>
27 #include <math.h>
28 #include <string>
29 #include "c_lib/rvgs.h"
30 #include "c_lib/rngs.h"
31
32 #define START      0.0          /* initial time */
33 #define STOP      20000.0       /* terminal (close the door) time */
34 // #define INFINITY (10000.0 * STOP) /* must be much larger than STOP */
35 #define MAXQUEUE   6          /* max. # of jobs in queue */
36
37
38 double Min(double a, double c)
39 /*
40 * return the smaller of a, b
41 */
42 {
43     if (a < c)
44         return (a);
45     else
46         return (c);
47 }
48
49
50 double GetArrival()
51 /*
52 * generate the next arrival time
53 */
54 {
55     static double arrival = START;
56
57     SelectStream(0);
58     arrival += Exponential(2.0);
59     return (arrival);
60 }
61
62
63
64 double GetService(double lb, double ub)
65 /*

```

```

66 * generate the next service time with rate 2/3
67 * -----
68 */
69 {
70     SelectStream(1);
71     return (Uniform(lb, ub));
72 }
73
74 /**
75 * bool checkArg()
76 *
77 * @param char* input – the input string literal from the console
78 * @return bool – true if the input is a number, false otherwise
79 *
80 * This function determines whether the argument is a number.
81 */
82
83 bool checkArg(char* input)
84 {
85     try
86     {
87         if (strlen(input) > 9)
88         {
89             throw std::logic_error("Number is too large.");
90         }
91
92         for (int i = 0; i < strlen(input); ++i)
93         {
94
95             if (std::isdigit(input[i]) || (input[i] == '.')) continue;
96             else
97             {
98                 std::string errorMessage;
99                 errorMessage.append((std::string)input);
100                 errorMessage.append(" is not a number.");
101                 throw std::logic_error(errorMessage);
102             }
103         }
104         return 1;
105     }
106
107     catch (const std::logic_error& error)
108     {
109         std::cerr << error.what() << std::endl;
110         return 0;
111     }
112 }
113
114 int main(int argc, char* argv[])
115 {
116     struct {
117         double arrival;           /* next arrival time
118     */
119         double completion;       /* next completion time
120     */
121         double current;          /* current time
122     */
123         double next;             /* next (most imminent) event time
124     */
125         double last;             /* last arrival time
126     */
127     } t;
128     struct {
129         double node;             /* time integrated number in the node
130     */

```

```

125     double queue;                /* time integrated number in the queue
126     */
126     double service;              /* time integrated number in service
127     */
127     } area = { 0.0, 0.0, 0.0 };
128     long index{};                 /* used to count departed jobs
129     */
129     long reject{};               /* number of jobs rejected because of full queue */
130     long number{};               /* number in the node
131     */
131     double endtime{};
132     int queuesize{};
133     double lowerBound{};
134     double upperBound{};
135     // Set the seed
136     for (int i = 0; i < argc; ++i)
137     {
138         if (*argv[i] == 's' && checkArg(argv[i + 1]))
139         {
140             PlantSeeds(std::stol(argv[i + 1]));
141             break;
142         }
143         else
144         {
145             PlantSeeds(123456789);
146         }
147     }
148
149     // Set the number of runs
150     for (int i = 0; i < argc; ++i)
151     {
152         if (*argv[i] == 'r' && checkArg(argv[i + 1]))
153         {
154             endtime = std::stol(argv[i + 1]);
155             break;
156         }
157         else
158         {
159             endtime = STOP;
160         }
161     }
162
163     // Set the queue size
164     for (int i = 0; i < argc; ++i)
165     {
166         if (*argv[i] == 'q' && checkArg(argv[i + 1]))
167         {
168             queuesize = std::stol(argv[i + 1]);
169             break;
170         }
171         else
172         {
173             queuesize = MAXQUEUE;
174         }
175     }
176
177     // Set the bounds for uniform distribution (service time)
178     for (int i = 0; i < argc; ++i)
179     {
180         if (*argv[i] == 'u' && checkArg(argv[i + 1]) && checkArg(argv[i + 2]))
181         {
182             lowerBound = std::stol(argv[i + 1]);
183             upperBound = std::stol(argv[i + 2]);
184             break;
185         }

```

```

186         else
187         {
188             lowerBound = 1.0;
189             upperBound = 2.0;
190         }
191     }
192     t.current = START;           /* set the clock */
193     t.arrival = GetArrival();    /* schedule the first arrival */
194     t.completion = INFINITY;    /* the first event can't be a completion */
195
196     while ((t.arrival < endtime) || (number > 0)) {
197         t.next = Min(t.arrival, t.completion);    /* next event time */
198         if (number > 0) {                        /* update integrals */
199             area.node += (t.next - t.current) * number;
200             area.queue += (t.next - t.current) * (number - 1);
201             area.service += (t.next - t.current);
202         }
203         t.current = t.next;                    /* advance the clock */
204
205         if (t.current == t.arrival) {          /* process an arrival */
206             if (number < queuesize)
207             {
208                 number++;
209                 t.arrival = GetArrival();
210                 if (t.arrival > endtime) {
211                     t.last = t.current;
212                     t.arrival = INFINITY;
213                 }
214                 if (number == 1)
215                     t.completion = t.current + GetService(lowerBound, upperBound);
216             }
217             else
218             {
219                 reject++;
220                 t.arrival = GetArrival();
221                 if (t.arrival > endtime) {
222                     t.last = t.current;
223                     t.arrival = INFINITY;
224                 }
225                 if (number == 1)
226                     t.completion = t.current + GetService(lowerBound, upperBound);
227             }
228         }
229
230         else {                                /* process a completion */
231             index++;
232             number--;
233             if (number > 0)
234                 t.completion = t.current + GetService(lowerBound, upperBound);
235             else
236                 t.completion = INFINITY;
237         }
238     }
239
240     printf("\nfor %ld jobs\n", index);
241     printf("    average interarrival time = %6.2f\n", t.last / index);
242     printf("    average wait ..... = %6.2f\n", area.node / index);
243     printf("    average delay ..... = %6.2f\n", area.queue / index);
244     printf("    average service time .... = %6.2f\n", area.service / index);
245     printf("    average # in the node ... = %6.2f\n", area.node / t.current);
246     printf("    average # in the queue .. = %6.2f\n", area.queue / t.current);
247     printf("    number of rejections..... = %6.2ld\n", reject);
248     printf("    percent of rejections.... = %6.2f\n", ((double)reject / (index+reject)) * 100);
249     printf("    utilization ..... = %6.2f\n", area.service / t.current);
250

```

```
251     return (0);  
252 }
```