UMass Boston CS 444
Project 4
Posted Monday, December 1, 2025
Due Friday, December 12, 2025 at 11:59pm

J.H.DeBlois

1 Overview

Proj4 builds alternative versions of a simulation of a queueing system, implemented as a C program named q.c. You will submit q1.c, q2.c, q3.c and q4.c. This proj4 description includes 7 sections:

- Getting Started
- Introduction
- Command Line Arguments
- Code Snippets Included
- Grading Rubric
- Appendix: Implementations

In your course directory, make a subdirectory named "proj4". Copy the supplied files to your course directory. The supplied files are in the instructor's cs444 subdirectory for proj4.

NOTE1: The supplied files must be in your proj4 directory.

Run `./simulationjhd` to start. Then explore the other files, each of which illustrates some feature you will need to understand (see more below).

NOTE2: The full code to modify will be posted on Thursday.

Incremental deliveries are required. Each time you work on your code on the server (or plan an upload), please copy your existing

```
qi.c, qi and readMe.txt to qi_<date>.c, qi_<date> and readMe_<date>.txt.
```

Then update the qi.c and readMe.txt and recompile to get your next qi. Please be sure you create at least two prior increments and leave them in your course directory.

NOTE3: Your first increment must be created at least 2 days before proj4 is due.

2 Getting Started

The simulator q.c is one process that contains several threads. Various components of the queueing system are simulated by the threads.

NOTE4: The simulation is a REAL-TIME system so the unit of time is ONE second.

1. You will be given C code to copy, modify, compile, test and submit. The simulator q.c makes use of the POSIX thread (pthread) library. The supplied files include hello32.c, which is the basic example from the Lawrence Livermore Lab Tutorial.

   NOTE5: On the server, to see the code, type: cat hello32.c and pipe to more.

2. Run the supplied compiled simulation with default input values by typing `./simulationjhd`.

   NOTE6: On the server, to run the simulation, type it now.

3 Introduction

The simulation relies on some background information.

1. Queueing theory was discussed in class. See queueing slides.

2. POSIX threads were discussed in class. See posixthreads slides.

3. Arrivals in the Queueing System. The arrival of customers follows an exponential distribution, which is characterized by a parameter λ. This λ is understood as the arrival rate.

   NOTE7: For example, if $\lambda = 8$, it means that eight customers will arrive per second on average.

4. Queue. An incoming customer enters the FIFO queue.

5. Servers in the Queueing System. The system has one or more servers. When a server is available, it fetches a customer from the queue and spends some amount of time to provide service to the customer. The service time is determined by another exponential distribution, characterized by μ, the service rate.

   NOTE8: For example, if $\mu = 10$, it means that a server can service ten customers per second on average.

6. Number of Servers. If there is only one server, a stable queueing system requires that λ be less than μ. Otherwise, customers arrive faster than service can be provided. If there are multiple servers, λ must be less than the number of servers times μ.

4 Command Line Arguments

Using `getopt.h` code, implement the following command line arguments:

- `-l lambda`: lambda is the arrival rate — default is 5.0
- `-m mu`: mu is the service rate — default is 7.0
- `-c numCustomer`: number of customers — default is 1000
- `-s numServer`: number of servers, between 1 and 5 — default is 1

NOTE7: The simulation code checks that `lambda` is less than `mu` times `numServer`
because otherwise the queueing system is unstable.

5 Code Snippets Included

This section explains useful snippets:

- typedef
- How to sleep to implement a time interval
- How to create reentrant code that can generate pseudorandom numbers
- How to draw from an exponential distribution
- How to implement the queue

5.1 typedef

The typedef specifier can alleviate the appearance of too many 'struct' words in your code. Look at supplied: grammar1_typedef_struct_customer_specifier.

5.2 How to Sleep to Implement a Time Interval

The classic Unix `sleep()` takes an integer argument and sleeps for that number of seconds. The threads in this project, however, need to sleep for fractional seconds. This can be done with `nanosleep()`. The following code shows how to sleep for 0.005 second.

```
#include <sys/time.h>

struct timespec sleepTime;

sleepTime.tv_sec = 0;
sleepTime.tv_nsec = 5000000L;

nanosleep(&sleepTime, NULL);
```

There are two time setup structs available in the include. One is struct timespec and the other is struct timeval. See the supplied examples: grammar2_struct_timeval_definition and grammar3_struct_timespec_definition.

5.3 How to Create Reentrant Code that can Generate Pseudorandom Numbers

The classic Unix rand() is poorly random. The newer drand48() has better randomness but is not reentrant — only one thread can run drand48() at a time. In this project, both threads 1 and 2 must generate pseudorandom numbers simultaneously and independently. You can use srand48_r() and drand48_r() for this purpose.

```
#include <stdlib.h>
#include <sys/time.h>

void *threadXYZ(void *p) {
struct drand48_data randData;
struct timeval tv;
double result;

gettimeofday(&tv, NULL);
//to seed the generator
srand48_r(tv.tv_sec + tv.tv_usec, &randData);

//to draw a number from [0, 1) uniformly and store it in "result"
drand48_r(&randData, &result);
}
```

The above code snippet should be present in both threads 1 and 2 so that they have local (private) sequences of pseudorandom numbers. These pseudorandom numbers are uniformly distributed between 0 and 1.

You have to seed the generator and use the drand48_data first. See supplied: grammar4_struct_drand46_data_definition. Now you have a random number from the uniform distribution. Next, use it to generate a random number from the exponential distribution.

5.4 How to Draw from an Exponential Distribution

Having generated a pseudorandom number between 0 and 1 uniformly, you can transform it to follow the exponential distribution with parameter λ as follows.

```
#include <math.h>

double rndExp(struct drand48_data *randData, double lambda) {
double tmp;

drand48_r(randData, &tmp);
return -log(1.0 - tmp) / lambda;
}
```

Note that the above code is reentrant as long as the private randData of a thread is passed as the parameter. Threads 1 and 2 can share this function. Therefore, only one copy of this function is needed.

5.5 How to Implement the Queue

The FIFO queue should be implemented as a linked list. A pthread mutex is used to coordinate exclusive access to the queue by the threads.

```
#include <pthread.h>
#include <sys/time.h>

typedef struct customer customer;
struct customer {
struct timeval arrivalTime;
customer *next;
};
customer *qHead, *qTail;
unsigned qLength;
pthread_mutex_t qMutex;
```

The variables qHead, qTail, qLength, and qMutex are global variables shared by all threads. Thread 1 inserts a new customer at the tail. Thread 2 removes a customer from the head.

```
void *thread1(void *p) {
customer *newCustomer;

loop
//draw inter-arrival time from exponential distribution
//sleep for that much time
newCustomer = (customer *) malloc(sizeof(customer));
//gettimeofday() timestamp the arrival time

/*lock qMutex
*consider 2 cases:
*1. insert into a nonempty queue
*2. insert into an empty queue -- signal thread 2
*unlock qMutex
*/
endLoop
}

void *thread2(void *p) {
customer *aCustomer;
struct timeval tv;
double waitingTime;

loop
/*lock qMutex
*consider 2 cases:
*1. a nonempty queue: remove from head
*2. an empty queue: wait for signal
*unlock qMutex
*/

gettimeofday(&tv, NULL); //timestamp the departure time
waitingTime = tv.tv_sec - aCustomer->arrivalTime.tv_sec +
(tv.tv_usec - aCustomer->arrivalTime.tv_usec) / 1000000.0;

/*draw service time from exponential distribution
*sleep for that much time
*/
free(aCustomer);
endLoop
}
```

When thread 1 inserts a new customer to an empty queue, it should signal thread 2 via a conditional variable. The above snippet also shows how to calculate the waiting time of a customer. Thread 3 can safely inspect the value of `qLength` without using the mutex.

6 Grading Rubric

The script will collect from you directory called `proj4` in the `cs444` folder. Put all files there, including the files you were given, your source code files, a Makefile, your executables called `q1, q2, q3, q4`, and `readMe.txt`. You must recompile your code on the server.

- (30 points) `readMe.txt`: Describe your high level design progress, any tough stretches that occurred in your work for q1 or q2 or q3 or q4, any outside sources that helped you including chatGPT. Make sure your code has the items in it that you talk about building in your readMe.
- (10 points) `Citing sources correctly in code`: If you include any code (copied or copied and modified) from an outside source, please follow the MIT guideline and show start and end markers for each section and for chatGPT generated code show the prompt. Keep in mind that we run MOSS to check similarity. You may be asked to come in and explain your code.
- (10 points) Incremental delivery was done as requested.
- (10 points) `Command line arguments`: It is required that you use `getopt.h` to create your command line arguments in your code.

- (20 points + optional 10 points XC) q1.c runs correctly `./q1` (same as `./q1 -l 5 -m 7 -c 300 -s 1 -d`). Do deletes to create No Server Available. Thread 2 is removed and only Thread 1 and Thread 3 run. You must actually delete lines of code - do not just comment them out. Add a debug option to show a comment about your work. For extra credit, add a display of interarrival times in 5 time bins.
- (10 points) q2.c runs correctly `./q2 -l 5 -m 7 -c 300 -s 1 -d`. Do deletes to create All Customrers Came Early. Thread 1 is removed and only Thread2 and Thread3 run. Put the actual number of customers in the queue at the start. Add a debug option to show a comment about your work.
- (10 points) q3.c runs correctly `./q3 -l 5 -m 7 -c 1000 -s 1 -d`. Visualize Server Utilization as a Timeline. All three threads run. Use code for additional thread(s) to create a timeline of 200 seconds to show when the server is busy serving customers - which is when thread2 is sleeping! Use extra threads and condition variables to collect the data needed to set some mark or space on the timeline. Add a debug option to show a comment about your work.
- (XC 10 points) q4.c runs correctly `./q4 -l 5 -m 2 -c 1000 -s 4 -d`. Allow each Server a Break. Everything runs as usual from the code you copy. There is additional code for additional thread(s) and condition variables to have each server stop working for a designated period of time but not concurrent with another server. Add a debug option to show a comment about your work.

You can run the reference executable `./simulationjhd -l 5 -m 8 -c 20 -s 1` to see the statistics generated by the code of the instructor. The observed results will change from run to run. You can also use the analytical formulas in the lecture notes on queueing theory to calculate theoretical prediction. For grading, the instructor or TA will run your code three times. You lose points if all three runs produce erroneous numbers that are more than thirty percent away from the reference numbers. The simulation is real-time. If λ is 5 and there are 1000 customers, the simulation is expected to take 200 seconds. During code development, you can get results faster by using fewer customers, such as `./q -c 200`.

7 Appendix: Implementations

Some discussion of implementation may be helpful.

7.1 Implement M/M/1 System

The arrival rate (λ) is 5 and the service rate (μ) is 7. There is one server. You write C code for the three threads that use the pthread library.

- Thread 1 generates the customers. It first draws a pseudorandom number t_c from the exponential distribution with parameter λ. This number t_c is the arrival time of the next customer. To simulate this arrival event, thread 1 will sleep for t_c seconds. When it wakes up, it inserts the new customer into the queue. If the queue was empty before the insertion, thread 1 will signal thread 2 that there is a customer now. Thread 1 repeats this process: draw t_c, sleep for t_c seconds, wake up, insert the customer into the queue, and signal thread 2 if necessary.
- Thread 2 simulates the server. It checks the queue to see if there is a customer. If there is no customer, thread 2 waits for a signal from thread 1. If there is a customer, thread 2 removes the customer from the queue to provide service. The service time t_s is drawn from the exponential distribution with parameter μ. To simulate providing service to a customer, thread 2 will sleep for t_s seconds. When it wakes up, it has finished with the customer. So it checks the queue again for the next customer. Thread 2 repeats this process: check the queue, if the queue is empty, go to sleep and wait for a signal, else remove a customer from the queue, draw t_s, and sleep for t seconds.
- Thread 3 observes the queue length. It repeats this process: record the queue length, and sleep for 0.005 seconds.

The main program initializes the three threads and waits for them to finish. To reach the steady state of this M/M/1 Markov process, we will run the simulation for 1,000 customers. Thus, thread 2 can terminate as soon as it has finished the 1,000th customers. However, threads 1 and 3 need to keep going until the 1,000th customer has left the system. The thread 1 keeps generating customers to maintain the steady state, and thread 3 keeps observing the queue length. You need to find a way to tell threads 1 and 3 to stop after thread 2 has stopped. After all threads have stopped and joined the main program, your code should print the following statistics:

- 1. The mean and standard deviation of inter-arrival time
- 2. The mean and standard deviation of waiting time
- 3. The mean and standard deviation of service time
- 4. The mean and standard deviation of queue length
- 5. Server utilization, which is busy time divided by total time

7.2 Implement Statistics: Online Average and Standard Deviation and Server Utilization

It is straightforward to calculate the average and standard deviation of the numbers in an array. However, there are situations when the numbers are coming in one by one, but we do not know how many will eventually come. We cannot save them in an array to be analyzed later when we do not have the length of the array beforehand. Therefore, we must do the calculation in an online fashion. That is, we calculate avg/std cumulatively without saving the individual numbers. See the code in `onlineAvgStd.c`. This should take care the first four statistics.

- 1. The mean and standard deviation of inter-arrival time
- 2. The mean and standard deviation of waiting time
- 3. The mean and standard deviation of service time
- 4. The mean and standard deviation of queue length
- 5. Server utilization, which is busy time divided by total time

To get the last number, we can use two timestamps, one at the beginning of the simulation and the other the end. Their difference is total time. Busy time is the sum of service time for all customers.

7.3 Implement M/M/N System

Expand the capabilities of the simulator by providing up to five servers. If the code for one server works well, it is a matter of spawning additional threads to also run the server code. The output of this part is the same as in the M/M/1 system, except that utilization is the average utilization of all servers.