UMass Boston CS 444
Project 4 Update: new Grading Rubric asks you to make changes to given code
Posted Sunday, April 20, 2025
Due Wednesday, April 30, 2025 at 11:59pm

1 Overview

Proj4 builds a simulation of a queueing system, implemented as a C program named q.c. **Use the ref.c code as the basis for your q.c and implement code for 4 experiments listed in the new Grading Rubric section below. All changes to this document are in bold font.** This proj4 description includes 7 sections:

- Introduction
- Command Line Arguments
- **New** Grading Rubric
- M/M/1 System
- **Online** Statistics
- M/M/n System
- Appendix A: Code Snippets

Since writing code is a process, please start by reading the proj4 description many times, until you understand it. Then sit and think about the overall design of your q.c.

Incremental deliveries are required. Each time you work on your code on the server (or plan an upload), please **use the day's date (or date and time) to copy your existing `q.c`, `q` and `readMe.txt` to `q.c_date`, `q_date` and `readMe.txt_date`. Then update the `q.c` and `readMe.txt` and recompile on the server to get `q`.** Please be sure you create at least two prior increments. Note: Your first increment must be created at least 2 days before proj4 is due. You also are required to copy the supplied files to your course directory.

2 Introduction

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

1. You write C code that makes use of the POSIX thread (pthread) library. The simulation is a REAL-TIME system so the unit of time is ONE second.
2. POSIX threads were discussed in class. See instructor website entry "posixthreads slides".

3. Queueing theory was discussed in class. See instructor website entry "queuing slides" (Note: both spellings queuing and queueing are correct!).

4. 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. For example, if $\lambda = 5$, it means that five customers will arrive per second on average. 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. For example, if $\mu = 7$, it means that a server can service seven customers per second on average.

   NOTE: 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.

   NOTE2: If there are multiple servers, λ must be less than the number of servers times μ.

3 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
- `-e1`**: experiment 1**
- `-e2`**: experiment 2**
- `-e3`**: experiment 3**
- `-e4`**: experiment 4**

NOTE3: Your code should check that `lambda` is less than `mu` times `numServer` because otherwise the queueing system is unstable.

4 New Grading Rubric

Create a directory called `proj4` in the `cs444` folder. Put all files there, including the files you were given, your source code, a Makefile, an executable called `q`, 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 **changes to threads, visuals or statistics and multiple servers** and any outside sources that helped you. Make sure your code has the items in it that you talk about in your readMe.
- (10 points) Command line arguments and citing sources: It is required that you use `getopt.h` to create your command line arguments **(5 debug switches have been added)** in your `q.c` and 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. **You do not have to mark any supplied code, but you do have to mark the start and end of the code for each of your changes.** 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.
- **(15 points) The M/M/1 system with Experiment 1 - No Server Available runs well using command `./q -l 5 -m 7 -c 1000 -s 1 -e1`. Thread 2 is removed and only Thread 1 and Thread 3 run. Delete the code for Thread 2, the code for displaying the queue and the code for statistics that require server information (you will lose points if the code is just commented out). Queue size will only increase, so visualizing the queue is not interesting; instead, explore specific inter-arrival times by adding a display of inter-arrival times for customers: Using the statistics on inter-arrival times, select 5 time bins (give your own definition of which range of interarrival times fall in each bin) and show in real-time how many inter-arrival times of your specified number of customers have fallen so far in each time bin. Put your code for this experiment in an if (e1) block so it only runs when switch e1 is specified on the command line. Your first statement in the block must be a printf statement that tells the user the gist of what you built for e1.**
- **(15 points) The M/M/1 system with Experiment 2 - All Customers Came Early runs well using command `./q -l 5 -m 7 -c 9 -s 1 -e2`. Thread 1 is removed and only Thread 2 and Thread 3 run. Delete all the code for Thread 1, but put the actual number of customers (input as 9 for the test) in the queue at the start. Also delete any statistic that requires customer arrival information. You will lose points if it is only commented out. The queue display will only decrease as customers are removed and served. Put your code for this experiment in an if (e2) block so it only runs when switch e2 is specified on the command line. As above, start with a printf statement that tells the user the gist of what you built for e2.**
- **(10 points) The M/M/1 system with Experiment 3 - Visualize Server Utilization As A Time-line runs well using command `./q -l 5 -m 7 -c 1000 -s 4 -e3`. The three threads, the queue visualization and the statistics from ref.c run as usual. There is additional code for additional thread(s) to build a timeline of 200 seconds to shows when the server is busy serving customers. This is actually when Thread 2 is sleeping. Use extra threads and condition variables to collect the data needed to set a mark (your choice of mark for server being used) or a space on the timeline for each of the 200 steps. Set your own criteria for what fractions of a second determine whether utilization is occurring in that second. Put your code for this experiment in an if (e3) block of statements. As above, start with a printf statement of the gist of what you built for e3.**
- **(10 points) The M/M/n system with Experiment 4 - Allow Each Server a Break runs well using command `./q -l 5 -m 7 -c 1000 -s 4 e4`. The three threads, queue visualization, statistics and the spawning for the multiple servers from ref.c run as usual. 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 concurrently when another server stops. Put your code for this experiment in an if (e4) block of statements. As above, start with a printf statement of the gist of what you built for e4.**

You can run the reference executable `/home/hdeblois/cs444/proj4/ref` 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`.

5 M/M/1 System

In the M/M/1 System, the arrival rate (λ) is 5 and the service rate (μ) is 7. There is one server. You **are given the M/M/1 System C code with 3 threads that use the pthread library. You are also given sample**

code in the instructor's directory `/home/hdeblois/cs444/proj4/*`.

- 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_s 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

5.1 Online 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.

6 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 just a matter of spawning additional threads to run the server code. The output of this part is the same as in M/M/1, except that utilization is the average utilization of all servers.

7 Appendix A: Code Snippets

This section explains useful snippets:

- 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

7.1 How to Sleep to Implement a Time Interval

An important aspect of the project is to sleep well. 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);
```

7.2 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.

7.3 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.

7.4 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.