UMass Boston CS 444
Project 4
Posted Tuesday, 21 April 2026
Due Tuesday, 14 May 2026 at 11:59pm

1 Introduction

This project implements a simulator of a queueing system. The simulator is one process that contains several threads. Various components of the queueing system are simulated by the threads. 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.

Queueing theory was discussed in class on April 7th. Briefly, 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.

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. 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.

2 Implement an 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. You can copy sample code that uses pthread from 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 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

2.1 How to Sleep

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);
```

2.2 Reentrant Code to 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.

2.3 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.

2.4 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.

2.5 Online Average and Standard Deviation

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.

3 Multiple Servers M/M/n

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.

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

Your code should check that λ `<` μ `x numserver` because otherwise the queueing system is unstable.

5 Grading Rubric

Create a directory called `proj4` in the `cs444` folder. Put all files there, including the source code `q.c`, a Makefile, an executable called `q`, and `readMe.txt`. You must recompile your code on the server.

You must upload a version of your `readMe.txt` 48 hours ahead of the duedate/duetime. It may not be AI-generated. It must explain steps taken and difficulties encountered and overcome. It must also include citations for sources and LLM prompts as explained in the syllabus.

You must upload a version of your code `q.c` 48 hours ahead of the duedate/duetime. This is your first incremental. You must compile it on the server and it must run.

Before you work again, please copy (use cp -p) both of the above to the appropriate `filename_date.filedesignator` so you keep the copies. Continue your work in the given file names.

Be sure your citations and prompts in your code are a subset of those in your `readMe.txt`.

- (20 points) `readMe.txt`: provide sample output, describe bugs or special features
- (20 points) command line arguments: it is required that you use `getopt.h`
- (60 points) the M/M/1 system `./q` (same as `./q -l 5 -m 7 -c 1000 -s 1`)
- Extra credit (20 points) the M/M/n system `./q -l 5 -m 2 -c 1000 -s 4`

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`.