Spring 2009

CSCI 551

Warm-up Project #2

Multi-threading

Due 11:45PM 2/13/2009

(Please check out the FAQ before sending your questions to the TA or the instructor.)

Assignment

In this assignment, you will emulate a M/M/2 queue depicted below using multi-threading within a single process. If you are not familiar with pthreads, you should read the *Pthreads Programming* book mentioned in the recommanded textbook section of the Course Description.

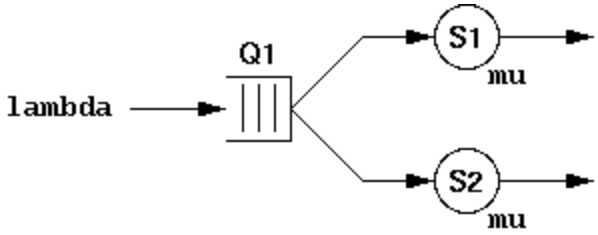


Figure 1: An M/M/2 queue.

There is a single stream of customers arriving to the system according to a Poisson process with rate `lambda`. (The inter-arrival time of a Poisson process with rate `lambda` is Exponentially distributed with mean `1/lambda`.) Customers are queued at Q1 and are served in first-in-first-out order. As server S1 or S2 becomes idle, it removes the customer at the head of the queue and begins to serve the customer. The service time of a customer is Exponentially distributed with mean `1/mu`. The unit for both `lambda` and `mu` is `(1/sec)`.

Please emulate the arrival process, S1, and S2, each with a thread. Make sure you protect Q1 with a mutex.

Compiling

Please use a `Makefile` so that when the grader simply enters:

```
make
```

an executable named `mm2` is created.

Please make sure that your submission conforms to other general compilation requirements and README requirements.

Commandline

The command line syntax for `mm2` is as follows:

```
mm2 [-lambda lambda] [-mu mu] [-s] \
    [-seed seedval] [-size sz] \
    [-n num] [-d {exp|det}] [-t tsfile]
```

Square bracketed items are optional. You must follow the UNIX convention that **commandline options** can come in any order. (Note: a **commandline option** is a commandline argument that begins with a **-** character in a commandline syntax specification.) Unless otherwise specified, output of your program must go to `stdout` and error messages must go to `stderr`.

The default value for `lambda` is 0.5 (customers per second) and the default value for `mu` is 0.35 (customers per second).

If the **`-s`** option is specified, only a single server should be present (server S2 is considered turned off). Then you have a M/M/1 system instead of the default M/M/2 system.

The **`-seed`** option specifies the value of the `l_seed` parameter to be used for the `InitRandom()` function below. `<seedval>` should be a non-negative long integer. The default value is 0.

The **`-size`** option specifies the size of Q1. The default value is 5.

The **`-n`** option specifies the total number of customers to arrive. The default value is 20.

The **`-d`** option specifies the probability distribution to use. By default, **`-d exp`** is assumed. In this mode, customers arrive according to a Poisson process with rate `lambda` and service times are Exponentially distributed with mean `1/mu`. If **`-d det`** is specified, inter-arrival times of customers are fixed and equal to `1/lambda` and their service times are also fixed and equal to `1/mu`.

If the **-t** option is specified, `tsfile` is a trace specification file that you should use to drive your simulation. In this case, you should ignore the **-lambda**, **-mu**, **-seed**, **-n**, and **-d** commandline options. You may assume that `tsfile` conforms to the format specification. (This means that if you detect a real error in this file, you may simply print an error message and call `exit()`.)

Running Your Code

The emulation should go as follows. At time 0, the arrival thread starts and both servers are idle. The first customer `c1` arrives at time `t1` (the 2nd customer `c2` arrives at time `t2`, and so on). The customers are queued at Q1. Since the queue is empty at time `t1`, the queue wakes up all idling servers and lets them compete for the customer at the head of the queue. If a server finds Q1 to be empty when it is woken up, it blocks. This process continues until all customers are served.

You are required to produce a detailed trace as the customers move through the system. The output **format** of your program **must** look like the following (please note that the values used here are just a bunch of unrelated random numbers for illustration purposes):

```
Parameters:
    lambda = 0.15         (if -t is not specified)
    mu = 0.35             (if -t is not specified)
    system = M/M/2   (or M/M/1 if -s is specified)
    seed = 0              (if -t is not specified)
    size = 5
    number = 20
    distribution = exp    (if -t is not specified)
    tsfile = FILENAME         (if -t is specified)

00000000.000ms: emulation begins
00000312.112ms: c1 arrives, inter-arrival time = 312.112ms
00000312.117ms: c1 enters Q1
00000312.119ms: c1 leaves Q1, time in Q1 = 0.002ms
00000312.120ms: c1 begin service
00000625.541ms: c2 arrives, inter-arrival time = 313.429ms
00000625.543ms: c2 enters Q1
00000772.497ms: c1 departs, service time = 460.377ms
                time in system = 460.385ms
00000772.502ms: c2 leaves Q1, time in Q1 = 146.959ms
00000772.504ms: c2 begin service
00000953.672ms: c3 arrives, inter-arrival time = 328.131ms
00000953.674ms: c3 enters Q1
00001217.773ms: c2 departs, service time = 445.269ms
```

```
                time in system = 592.232ms
...
????????.???ms: c20 departs, service time = ???.???ms
                time in system = ???.???ms

Statistics:

    average inter-arrival time = <real-value>
    average service time = <real-value>

    average number of customers in Q1 = <real-value>
    average number of customers at S1 = <real-value>
    average number of customers at S2 = <real-value>

    average time spent in system = <real-value>
    standard deviation for time spent in system = <real-value>

    customer drop probability = <real-value>
```

In the `"begin service"` and the `"departs"` trace output, you may optinally include the name of the server at which the customer is served. For example, you may output `"c1 begin service at S1"` and `"c1 departs from S1, ..."`.

If a customer *n* is dropped because of buffer overflow, you should print:

```
????????.???ms: cn dropped
```

where the question marks was the time customer *n* was dropped.

Please note that each departure line in the above trace have been broken up into 2 lines. This is done because of the way web browsers handle pre-formatted text. Please print them all in one line in your program.

All real values in the statistics must be printed with at least 6 significant digits. (If you are using `printf()`, you can use `"%.6g"`.) A timestamp in the beginning of a line of trace output must be in milliseconds with 8 digits (zero-padded) before the decimal point and 3 digits (zero-padded) after the decimal point.

Please use sample means when you calculated the averages. If *n* is the number of sample, this mean that you should divide things by *n* (and not *n-1*).

The unit for time related *statistics* must be in seconds.

Let X be something you measure. The standard deviation of X is the square root of the variance of X. The variance of X is the average of the square of X minus the square of the average of X. Let E(X) denote the average of X, you can write:

$$Var(X) = E(X^2) - [E(X)]^2$$

If the user presses <Cntrl>c on the keyboard on send a SIGINT or SIGTERM to your simulation process, you must stop the arrival process, remove all customers in Q1, let your system finish serving all the customers in the usual way, and output statistics in the usual way. (Please note that it may not be possible to remove all customers in Q1 at the time of the interrupt. The idea here is that once the interrupt has occurred, the only customers you should server are the only onces in service. All other customers should be removed from the system.)

Random Numbers Generation

In order to make grading easier, you are required to generate random numbers in a fixed way. First you must seed the random number generator with a user-specified value using the `InitRandom()` function specified below:

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

#define round(X) (((X)>= 0)?(int)((X)+0.5):(int)((X)-0.5))

void InitRandom(long l_seed)
{
    if (l_seed == 0L) {
        time_t localtime=(time_t)0;

        time(&localtime);
        srand48((long)localtime);
    } else {
        srand48(l_seed);
    }
}

int GetInterval(int exponential, double rate)
{
    if (exponential) {
        double dval=(double)drand48();
```

```
            return ExponentialInterval(dval, rate);
        } else {
            double millisecond=((double)1000)/rate;

            return round(millisecond);
        }
    }
```

You need to write the `ExponentialInterval(double dval, double rate)` function where:

- `dval` is a value in the interval [0.0, 1.0)
- `rate` has unit 1/sec

The return value of `GetInterval()` should be an integer having a unit of milliseconds. To make things easier to manage, we limit the value of this integer to be between 1 and 10,000. This means that before `GetInterval()` returns, you must examine the value it plans to return. If it's too small, set it to 1, if it's too large, set it to 10,000. This also applies to the values you read from `tsfile`.

Your program should call `InitRandom()` exactly once before it ever calls `GetInterval()`. A good place to call these function is right after you have done parsing the commandline options.

Trace Specification File Format

The trace specification file is an ASCII file containing **n+1** lines (each line is terminated with a "\n") where **n** is the total number of customers to arrive. Line 1 of the file contains an integer which corresponds to the value of **n**. Line **k** of the file contains the inter-arrival time and service time (separated by space or tab characters), in milliseconds, for customer **k-1**. A line may contain leading and trailing space or tab characters. A sample tsfile for **n=3** customers is provided. It's content is listed below:

```
3
2716 16253
7721 35149
972  2614
```

In the above example, customer 1 is to arrive 2716ms after simulation starts and its service time should be 16253ms; the inter-arrival time between customer 2 and 1 is to be 7721ms and the service time of customer 2 should be 35149ms; the inter-arrival time between customer 3 and 2 is to be 972ms and the service time of customer 3 should be 2614ms.

You may assume that this file is error-free. (This means that if you detect a real error in this file, you may simply print an error message and call `exit()`.)

Grading Guidelines

The grading guidelines has been made available. Please run the scripts in the guidelines on `nunki.usc.edu`. It is possible that there are bugs in the guidelines. If you find bugs, please let the instructor know as soon as possible.

Please note that although the grader will follow the grading guidelines to grade, the grader may use a different set of trace files and commandline arguments.

Miscellaneous Requirements & Hints

- You are required to use *separate compilation* to compile your source code. You must divide your source code into separate source files in a logical way. You also must **not** put the bulk of your code in header files!

- Please use `gettimeofday()` to get time information with a microsecond resolution. You can use `select()` or `usleep()` (or equivalent) to sleep for a specified number of microseconds.

- Your must not do busy-wait! If you run `"top"` from the commandline and you see that your program is taking up one of the top spots in CPU percentages and show high percentages (more than 1%), this is considered **busy-wait**. You will lose a lot of points if your program does busy-waiting.

  It's quite easy to find the offending code. If you run your program from the debugger, wait a few seconds, then type <Cntrl+C>. Most likely, your program will break inside your busy-waiting loop! An easy fix is to call `select()` to sleep for 100 millisecond before you loop again.

- Please round inter-arrival and service times to the nearest millisecond using the `round()` macro above.

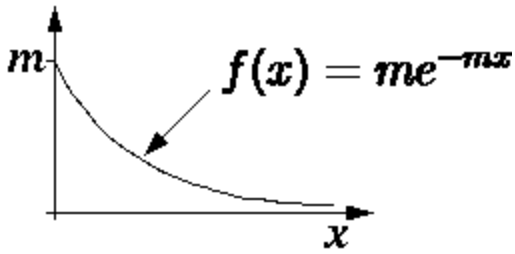
- If you have `.nfs*` files you cannot remove, please see notes on `.nfs` files.

Here are some additional hints:

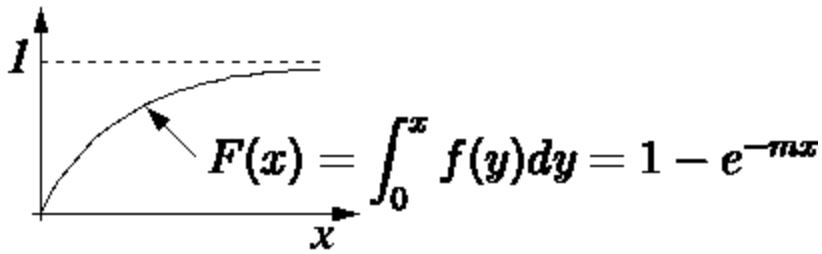
- For this assignment, you should implement a **time-driven** emulation (and *not* an event-driven simluation such as ns-2). For an event-driven simulaiton, you can easily implement this project using a single thread and an event queue. In a time-driven emulation, a thread must sleep for the amount of time that it supposes to take to do the a job. For example, if a server would take 317 milliseconds to serve a job, it would actually sleep (using `select()`) for some period of time so that the packet seem to stay in the server for 317 milliseconds. Similarly, if the arrival process needs to wait 634 milliseconds between the arrivals of customers `c1` and `c2`, it should sleep for some period of time so that it looks like customer `c2` arrives 634 milliseconds after customer `c1` has arrived.

- You need to calculate the time correctly for the `select()` mentioned above. For example, with Poisson arrivals, let's say that the arrival process needs to wait 634 milliseconds between the arrivals of customers `c1` and `c2`. Assuming that it took 45 milliseconds to do bookkeeping and to enqueue the customer to the queueing system, you should sleep for 589 milliseconds (and not sleep for 634 milliseconds). Please note the bookkeeping time for servers **should be** zero!

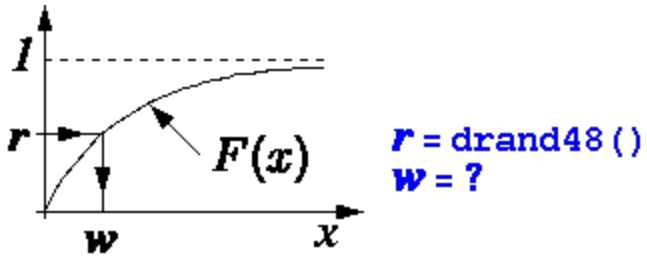
- Let *f(x)* denote the Probability Density Function (pdf) of an Exponential Distribution with rate m. Pictorially,



Let *F(x)* be the corresponding Probability Distsribution Function (PDF) of *f(x)*. Pictorially,



To randomly choose a value *w* according to distribution *f(x)*, you can pick a random value *r* from a uniform distribution between 0 and 1 and map *r* across *F(x)* to get *w*. This procedure is depicted below.



Please note that this procedure works in general for any

function *f(x)* and the corresponding *F(x)*.

Hint: `w = ln(1-r) / (-m)` where `ln()` is the natural logarithm function.

[*Last updated Thu Feb 12 2009*] [*Please see copyright regarding copying.*]