Summer 2013

CSCI 402

Warmup Assignment #2

(100 points total)

Multi-threading - Token Bucket Emulation in C

Due 11:45PM 6/18/2013 (firm)

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

Assignment Description

In this assignment, you will emulate/simulate a **traffic shaper** who transmits packets controlled by a **token bucket filter** depicted below using multi-threading within a single process. If you are not familiar with pthreads, you should read Chapter 2 of our required textbook.

IMPORTANT: Please note that this assignment is posted before all the components (such as **thread creation and joining**, **signal handling**, and **guarded commands**) you need to implement this project have been covered in lectures. If you do **not** want to learn about these components on your own (by learning from the textbook), please delay starting this project until they are covered in class. There will be plenty of time to implement this project after the relevant topics are covered in class.

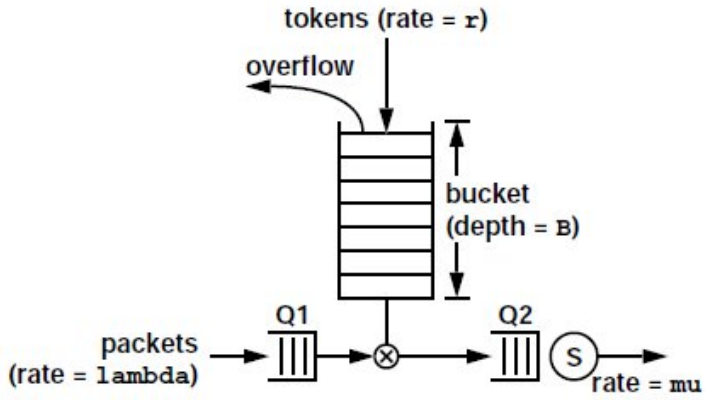

Figure 1: A system with a token bucket filter.

Figure 1 above depicts the **system** you are required to emulate. The **token bucket** has a capacity (bucket depth) of `B` tokens. Tokens arrive into the token bucket at a **constant rate** of `r` tokens per second. Extra tokens (overflow) would simply disappear if the token bucket is full. A token bucket, together with its control mechanism, is referred as a **token bucket filter**.

Packets arrive at the token bucket filter at a rate of `lambda` packets per second (i.e., packets have an inter-arrival time of `1/lambda`) and each packet requires `P` tokens in order for it to be eligiable for transmission. (Packets that are eligiable for transmission are queued at the `Q2` facility.) When a packet arrives, if `Q1` is not empty, it will just get queued onto the `Q1` facility. Otherwise, it will check if the token bucket has `P` or more tokens in it. If the token bucket has `P` or more tokens in it, `P` tokens will be removed from the token bucket and the packet will join the `Q2` facility (technically speaking, you are **required** to first add the packet to `Q1` and timestamp the packet, remove the `P` tokens from the token bucket and the packet from `Q1` and timestamp the packet, before moving the packet into `Q2`). If the token bucket does not have enough tokens, the packet gets queued into the `Q1` facility. You should also check if there is enough tokens in the bucket so you can move the packet at the head of `Q1` into `Q2`.

The transmission facility **S** serves packets in **Q2** at a service rate of **mu** per second. When a packet has received **1/mu** seconds of service, it leaves our system.

When a token arrives at the **token bucket**, it will add a token into the **token bucket**. If the bucket is already full, the token will be lost. It will then check to see if **Q1** is empty. If **Q1** is not empty, it will see if there is enough tokens to make the packet at the head of **Q1** be eligiable for transmissions. If it does, it will remove the corresponding number of tokens from the token bucket, remove that packet from **Q1** and move it into **Q2**, and **wake up** the server (by broadcasting the corresponding condition). Technically speaking, the "server" is not part of the token bucket filter. Nevertheless, it's part of this assignment to emulation the sever as well.

Our system can run in only one of two modes.

Deterministic : In this mode, all inter-arrival times are equal to **1/lambda** seconds, all packets require exactly **P** tokens, and all service times are equal to **1/mu** seconds. If **1/lambda** is greater than 10 seconds, please use an inter-arrival time of 10 seconds. If **1/mu** is greater than 10 seconds, please use an service time of 10 seconds.

Trace-driven : In this mode, we will drive the emulation using a tracefile. Each line in the trace file specifies the **inter-arrival time** of a packet, the **number of tokens** it need in order for it to be eligiable for transmission, and its **service time**.

Your job is to emulate the packet and token arrivals, the operation of the token bucket filter, the first-come-first-served queues **Q1** and **Q2**, and server **S**. You also must produce a trace of your emulation for every important event occurred in your emulation. Please see more details below for the requirements.

You must use:

- one thread for packet arrival
- one thread for token arrival
- one thread for server

You must **not** use one thread for each packet.

In addition, you must use at least one mutex to protect **Q1**, **Q2**, and the token bucket.

Finally, **Q1** and **Q2** must have infinite capacity (i.e., you should use **My420List** from warmup assignment #1 to implement them and **not** use arrays).

We will **not** go over the lecture slides for this assignment in class. Although it's important that you are familiar with it. Please read it over. If you have questions, please e-mail the **instructor**.

Compiling

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

```
make warmup2
```

an executable named **warmup2** is created. Please make sure that your submission conforms to other general compilation requirements and README requirements.

Commandline

The command line syntax for **warmup2** is as follows:

```
warmup2 [-lambda lambda] [-mu mu] \
    [-r r] [-B B] [-P P] [-n num] \
    [-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 `lambda`, `mu`, `r`, `B`, and `P` parameters all have obvious meanings. The **`-n`** option specifies the total number of packets to arrive. If the **`-t`** option is specified, `tsfile` is a trace specification file that you should use to drive your emulation. In this case, you should ignore the **`-lambda`**, **`-mu`**, **`-P`**, and **`-num`** commandline options and run your emulation in the trace-driven mode. You may assume that `tsfile` conforms to the tracefile format specification. (This means that if you detect an error in this file, you may simply print an error message and call `exit()`. There is no need to perform error recovery.) If the **`-t`** option is not used, you should run your emulation in the deterministic mode.

The default value (i.e., if it's not specified in a commandline option) for `lambda` is 0.5 (packets per second), the default value for `mu` is 0.35 (packets per second), the default value for `r` is 1.5 (tokens per second), the default value for `B` is 10 (tokens), the default value for `P` is 3 (tokens), and the default value for `num` is 20 (packets). `B`, `P`, and `num` must be positive integers with a maximum value of 2147483647 (0x7fffffff). `lambda`, `mu`, and `r` must be positive real numbers.

If **`1/r`** is greater than 10 seconds, please use an inter-token-arrival time of 10 seconds.

Running Your Code and Program Output

The emulation should go as follows. At emulation time 0, all 3 threads (arrival, token depositing, and server threads) got started. The arrival thread would sleep so that it can wake up at a time such that the inter-arrival time of the first packet would match the specification (either according to **`lambda`** or the first record in a tracefile). At the same time, the token depositing thread would sleep so that it can wake up every **`1/r`** seconds and would try to deposit one token into the token bucket. The first packet `p1` arrives at time `t1` (the 2nd packet `p2` arrives at time `t2`, and so on).

As a packet arrives, you need to follow the operational rules of the token bucket filter and determine if the packet should be queued onto **`Q1`** (and no tokens should be removed) or **`Q2`** (and the correct number of tokens removed). There is one exception to the rules though. If the number of tokens required by a packet is larget than the bucket depth, the packet must be dropped (otherwise, it will block all other packets that follow it). If the packet is to be queued onto **`Q2`**, when the mutex is locked, the arrival thread should broadcast the condition to wake up the potentially sleeping server thread.

As a token arrives, you also need to follow the operational rules of the token bucket filter and determine if the token should be added to the token bucket filter or not. You should also check for the condition where **`Q1`** is not empty and there are enough tokens in the token bucket. If this is the case, you should remove the correct number of tokens from the token bucket and move the packet at the head of **`Q1`** into **`Q2`**. When this thread has the mutex locked, it should broadcast the condition to wake up the potentially sleeping server thread.

As a server thread wakes up, it should lock the mutex and check if **`Q2`** is empty. If it's not empty, it should remove the first packet in **`Q2`** and sleep for the service time of that packet. If **`Q2`** is empty, it should get blocked waiting for the condition variable (and release the mutex simultaneously).

You are required to produce a detailed trace as the packets move through the system. The output **format** of your program **must** satisfy the following requirements.

- You must first print all the emulation paramters. Please see the sample printout for what the output must look like.
- Whenever a token arrives, you must assign a number to it, and add it to the token bucket. You must then print its arrival time, the fact that it has arrived, and the number of tokens in the the token bucket. Please see the sample printout for what the output must look like.
- Whenever a packet arrives, you must assign a number to it. You must then print its arrival time, the fact that it has arrived, the number of tokens it needs for transmission, and the time between its arrival time and the arrival time of the previous packet. Please see the sample printout for what the output must look like.

  You then must append this packet onto `Q1`. Afterwards, you must then print the time this packet entered `Q1` and the fact that it has entered `Q1`. Please see the sample printout for what the output must look like.

  Later on, when this packet leaves `Q1`, it removes the correct number of tokens from the token bucket. You must then print the time this packet leaves `Q1`, the fact that it has left `Q1`, the amount of time it spent in `Q1`, and the number of tokens in the the token bucket. Please see the sample printout for what the output must look like.

  You must then append this packet onto `Q2`. Afterwards, you must then print the time this packet entered `Q2` and the fact that it has entered `Q2`. Please see the sample printout for what the output must look like.

Later on, when this packet leaves `Q2` and enters the server, you must then print the time this packet begin service, the fact that it has begun service, and the amount of time it spent in `Q2`. Please see the sample printout for what the output must look like.

- When emulation ends, you must print all the necessary statistics. Please see the sample printout for what the output must look like. If a particular statistics is not applicable (e.g., will cause divide-by-zero error), instead of printing a numeric value, please print "N/A" followed by an explanation (such as "no packet arrived at this facility").

Below is an example what your program output must look like (please note that the values used here are just a bunch of unrelated random numbers for illustration purposes):

```
Emulation Parameters:
    lambda = 0.5          (if -t is not specified)
    mu = 0.35             (if -t is not specified)
    r = 1.5
    B = 10
    P = 3                 (if -t is not specified)
    number to arrive = 20 (if -t is not specified)
    tsfile = FILENAME         (if -t is specified)

00000000.000ms: emulation begins
00000251.726ms: token t1 arrives, token bucket now has 1 token
00000502.031ms: token t2 arrives, token bucket now has 2 tokens
00000503.112ms: p1 arrives, needs 3 tokens, inter-arrival time = 503.112ms
00000503.376ms: p1 enters Q1
00000751.148ms: token t3 arrives, token bucket now has 3 tokens
00000751.186ms: p1 leaves Q1, time in Q1 = 247.810ms, token bucket now has 0 token
00000752.716ms: p1 enters Q2
00000752.932ms: p1 begin service at S, time in Q2 = 0.216ms
00001004.271ms: p2 arrives, needs 3 tokens, inter-arrival time = 501.159ms
00001004.526ms: p2 enters Q1
00001007.615ms: token t4 arrives, token bucket now has 1 token
00001251.259ms: token t5 arrives, token bucket now has 2 tokens
00001505.986ms: p3 arrives, needs 3 tokens, inter-arrival time = 501.715ms
00001506.713ms: p3 enters Q1
00001507.552ms: token t6 arrives, token bucket now has 3 tokens
00001508.281ms: p2 leaves Q1, time in Q1 = 503.755ms, token bucket now has 0 token
00001508.761ms: p2 enters Q2
...
00003612.843ms: p1 departs from S, service time = 2859.911ms, time in system = 3109.731ms
00003613.504ms: p2 begin service at S, time in Q2 = 2104.743ms
...
????????.???ms: p20 departs from S, service time = ???.???ms, time in system = ???.???ms

Statistics:

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

    average number of packets in Q1 = <real-value>
    average number of packets in Q2 = <real-value>
    average number of packets at S = <real-value>

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

    token drop probability = <real-value>
    packet drop probability = <real-value>
```

The first column contains timestamps and they correspond to event times, measured relative to the start of the emulation. Please use 8 digits to the left of the decimal point and 3 digits after the decimal point for all the timestamps in this column. All time intervals must be printed in milliseconds with 3 digits after the decimal point. In the printout, after emulation parameters, all values reported must be **measured** values.

The **average number of packets** at a facility can be obtained by adding up all the time spent at that facility (for all packets) divided by the total emulation time. The **time spent in system** for a packet is the difference between the time the packet departed from the server and the time that packet arrived. The **token drop probability** is the total number of tokens dropped because the token bucket was full divided by the total number of tokens that was produced by the token

dropped because the token bucket was full divided by the total number of tokens that was produced by the token depositing thread. The **packet drop probability** is the total number of packets dropped because the number of tokens required is larger than the bucket depth divided by the total number of packets that was produced by the arrival thread.

If token *n* is dropped because of token bucket overflow, you must print:

```
????????.???ms: token tn arrives, dropped
```

where the question marks was the time token *n* arrived (and dropped). Similarlily, if packet *n* is dropped because the number of token it needs is larger than the bucket depth, you must print:

```
????????.???ms: packet pn arrives, needs ? tokens, dropped
```

where the question marks was the time packet *n* arrived (and 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 (and not milliseconds).

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:

$$\mathrm{Var}(X) = E(X^2) - [E(X)]^2$$

If the user presses <Cntrl+C> on the keyboard, you must stop the arrival thread and the token depositing thread, remove all packets in **`Q1`** and **`Q2`**, let your server finish serving the current packet in the usual way, and output statistics in the usual way. (Please note that it may not be possible to remove all packets in **`Q1`** at the instance of the interrupt. The idea here is that once the interrupt has occurred, the only packets you should server are the only one in service. All other packets should be removed from the system.)

Finally, when no more packet can arrive into the system, you must stop the arrival thread as soon as possible. Also, when Q1 is empty and no future packet can arrival into Q1, you must stop the token depositing thread as soon as possible.

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 packets 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 in milliseconds (an integer), the number of tokens required (an integer), and service time in milliseconds (an integer) for packet **`k-1`**. The 3 fields are separated by space or tab characters. There must be no leading or trailing space or tab characters in a line. A sample tsfile for **`n=3`** packets is provided. It's content is listed below:

```
3
2716    2    9253
7721    1   15149
972     3    2614
```

In the above example, packet 1 is to arrive 2716ms after emulation starts, it needs 2 tokens to be eligible for transmission, and its service time should be 9253ms; the inter-arrival time between packet 2 and 1 is to be 7721ms, it needs 1 token to be eligible for transmission, and its service time should be 15149ms; the inter-arrival time between packet 3 and 2 is to be 972ms, it needs 3 token to be eligible for transmission, and its service time 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 must simply print an error message and call `exit()`. There is no need to perform error recovery.)

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. (**Note:** the grading guidelines is subject to change without notice.)

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.

The grading guidelines is the **only** grading procedure we will use to grade your program. No other grading procedure will be used. Please note that the grader may use a different set of trace files and commandline arguments. (We may also make minor changes if we discover bugs in the script or things that we forgot to test.) It is strongly recommended that you run your code through the scripts in the grading guidelines.

Miscellaneous Requirements & Hints

- Please read the programming FAQ if you need a refresher on file I/O and bit/byte manipulications in C.
- You must **NOT use any external code segments** to implement this assignment. You must implement all these functionalities from scratch.
- 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/usleep()` to sleep for 100 millisecond before you loop again.
- If you have `.nfs*` files you cannot remove, please see notes on `.nfs` files.

Here are some additional hints:

- For this assignment, you are implementing a **time-driven** emulation (and *not* an event-driven simluation such as ns-2). For an event-driven emulation, 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/usleep()`) for some period of time so that the packet seem to stay in the server for 317 milliseconds. Similarly, if the arrival thread needs to wait 634 milliseconds between the arrivals of packets `p1` and `p2`, it should sleep for some period of time so that it looks like packet `p2` arrives 634 milliseconds after packet `p1` has arrived.
- You need to calculate time correctly for the `select/usleep()` call mentioned above. For example, if the arrival thread needs to wait 634 milliseconds between the arrivals of packets `p1` and `p2`. Assuming that it took 45 milliseconds to do bookkeeping and to enqueue the packet to the queueing system, you should sleep for 589 milliseconds (and not sleep for 634 milliseconds). Please note such calculation does not apply to the server.

Submission

All assignments are to be submitted electronically - including your README file. To submit your work, you must first `tar` all the files you want to submit into a **tarball** and `gzip` it to create a **gzipped tarfile** named **`warmup2.tar.gz`**. Then you upload **`warmup2.tar.gz`** to the Bistro system. On `nunki.usc.edu` or `aludra.usc.edu`, the command you can use to create a gzipped tarfile is:

```
/usr/usc/bin/gtar cvzf warmup2.tar.gz MYFILES
```

Where **`MYFILES`** is the list of file names that you are submitting (you can also use wildcard characters if you are sure that it will pick up only the right files). **DO NOT** submit your compiled code, just your source code and README file. **Two point will be deducted** if you submit extra binary files, such as `warmup2`, `.o`, `core`, or files that can be **generated** from the rest of your submission.

Please note that the 2nd commandline argument of the `gtar` command above is the **output** filename of the `gtar` command. So, if you omit `warmup2.tar.gz` above, you may accidentally replace one of your files with the output of the `gtar` command. So, please make sure that the first commandline argument is **`cvzf`** and the 2nd commandline argument is **`warmup2.tar.gz`**.

In your README file, you should include the command that the grader should use to compile your code to generate **`warmup2`**. If you don't include such instruction, the grader will assume that the command specified in the spec should be used to compile your code. But if they can't get your code to compile easily and you don't submit compilation instructions, you will lose points. Please also note that you MUST include a README file in your submission. If you have nothing to put inside the README file, please write "(This file intentionally left blank.)" in your README file.

Here is a sample command for creating your `warmup2.tar.gz` file (your command will vary depending on what files you want to submit):

```
/usr/usc/bin/gtar cvzf warmup2.tar.gz *.c *.h Makefile README
```

You should read the output of the above commands carefully to make sure that `warmup2.tar.gz` is created properly. If you don't understand the output of the above commands, you need to learn how to read it! It's your responsibility to ensure that `warmup2.tar.gz` is created properly.

You need to run **`bsubmit`** to submit warmup2.tar.gz to the submission server. Please use the following command:

```
~csci551b/bin/bsubmit upload \
    -email `whoami`@usc.edu \
    -event bourbon.usc.edu_80_1291227186_95 \
    -file warmup2.tar.gz
```

Please note that the quotation marks surrounding `whoami` are **back-quote** characters and not single quotes. It's best if you just copy and paste the above command into your console and not try to type the whole command in.

If the command is executed successfully, the output should look like the sample mentioned in the submission web page. If it doesn't look like that, please fix your command and rerun it until it looks right. If there are problems, please contact the instructor.

It is extreme important that you also **verify your submission** after you have submitted `warmup2.tar.gz` electronically to make sure that everything you have submitted is everything you wanted us to grade.

Finally, please be familiar with the Electronic Submission Guidelines and information on the bsubmit web page.

[*Last updated Wed Jun 05 2013*] [*Please see copyright regarding copying.*]