

## Warmup Assignment #2

(100 points total)

### Multi-threading - Token Bucket Emulation in C

*Due 11:45PM 9/30/2022 (firm)*

[ This content is protected and may not be shared, uploaded, or distributed. ]

This spec is **private** (i.e., only for students who took or are taking CSCI 402 at USC). You do **not** have permissions to **display this spec** at a public place (such as a public bitbucket/github). You also do **not** have permissions to **display the code** you write to implement this spec at a public place since your code was written to implement a private spec. (If a prospective employer asks you to post your code, please tell them that you do not have permissions to do so; but you can send them a **private copy**.)

To download the spec below in one command (so you can make a backup of the spec in case the class web server is not available due to network or server problem), do the following inside a terminal in Ubuntu 16.04 (preferable in an empty directory) to make a quick and dirty (may be incomplete) backup of our spec:

```
wget -r -l 1 --user=USERID --password=PASSWORD http://merlot.usc.edu/cs402-f22/projects/warmup2/index.html
```

where USERID and PASSWORD are the user ID and password used to access protected content from our class web site. Then type the following in the terminal:

```
firefox merlot.usc.edu/cs402-f22/projects/warmup2/index.html
```

Please remember that I do update the spec occasionally. You should **ONLY** look at your backup copy when the class web site is unavailable.

#### Assignment Description

(Please check out the [Warmup 2 FAQ](#) before sending your questions to the TAs, the course producers, or the instructor.)

In this assignment, you will emulate/simulate a **traffic shaper** that transmits/services 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 background materials (e.g., **Unix signals**) 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. If you don't want to wait and don't want to learn about Unix signals and stuff on your own, I would strongly recommend that you do this assignment in two steps. First, you write your code to do the emulation **without** any **<Ctrl+C>**-handling code. By the time you get this part to work perfectly, we would have covered everything you need to finish the assignment. Then you follow the recommendations in the lecture to add **<Ctrl+C>**-handling code.

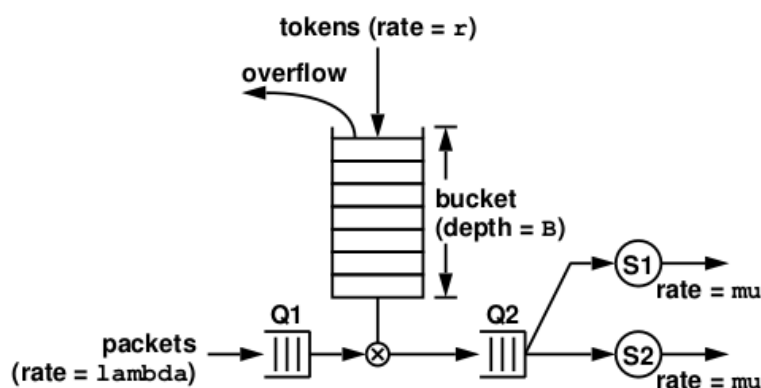


Figure 1: A system with a token bucket filter.

**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 according to an unusual arrival process where the inter-token-arrival time between two consecutive tokens is  $1/r$ . We will call **r** the **token arrival rate** (although technically speaking, it's not exactly the token arrival rate; please understand that this is **quite different** from saying that the tokens arrive at a constant rate of **r**). Extra tokens (overflow) would simply disappear if the token bucket is full. A token bucket, together with its control mechanism, is referred to as a **token bucket filter**.

Packets arrive at the token bucket filter according to an unusual arrival process where the **inter-arrival time** between two consecutive packets is  $1/\lambda$ . We will call  **$\lambda$**  the **packet arrival rate** (although technically speaking, it's not exactly the packet arrival rate; please understand that this is **quite different** from saying that the packets arrive at a constant rate of  **$\lambda$** ). Each packet requires **P** tokens in order for it to be eligible for transmission. (Packets that are eligible 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. (Please note that, **in this case**, you do **not** have to check if there is enough tokens in the bucket so you can move the packet at the head of **Q1** into **Q2** and you need to understand why you do **not** need to perform such a check.) 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**), and **wake up** the servers in case they are sleeping. If the token bucket does not have enough tokens, the packet gets queued into the **Q1** facility. Finally, if the number of tokens required by a packet is larger than the bucket depth, the packet must be **dropped** (otherwise, it will block all other packets that follow it).

The transmission facility (denoted as **S1** and **S2** in the above figure and they are referred to as the "**servers**") serves packets in **Q2** in the first-come-first-served order and at a transmission/service rate of  **$\mu$**  per second. When a server becomes available, it will dequeue the first packet from **Q2** and start transmitting/servicing the packet. When a packet has received  $1/\mu$  seconds of service, it leaves the system. You are required to keep the servers **as busy as possible**.

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 eligible for transmission (packets in **Q1** are also served in the first-come-first-served order). 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 servers in case they are sleeping. It will then check the packet that is now at the head of **Q1** to see if it's also eligible for transmission, and so on.

Technically speaking, the "servers" are not part of the "token bucket filter". Nevertheless, it's part of this assignment to emulation/simulation the servers because the servers are considered part of the "system" to be emulated. (For the purpose of this spec, we will use the term "emulation" and "simulation" interchangeably.)

Our system can run in only one of two modes.

**Deterministic** : In this mode, all **inter-arrival times** are equal to  $1/\lambda$  seconds and all service times are equal to  $1/\mu$  seconds (**both these values must be rounded to the nearest millisecond**), and all packets require exactly **P** tokens. 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 a service time of 10 seconds.

**Trace-driven** : In this mode, we will drive the emulation using a **trace specification file** (will be referred to as a "**tsfile**"). 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 eligible for transmission, and its **service time**. (Please note that in this mode, it's perfectly fine if an inter-arrival time or a service time is greater than 10 seconds.) If you are running in the trace-drive mode, you must **not** validate or read the entire **tsfile** before you start your simulation.

In either mode, if  $1/r$  is greater than 10 seconds, please use an inter-token-arrival time of 10 seconds. Otherwise, please **round** the inter-token-arrival time to the nearest millisecond.

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 servers **S1** and **S2**. 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 each 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. (It is recommended that you use **exactly one** mutex to protect **Q1**, **Q2**, and the token bucket.)

Finally, **Q1** and **Q2** must have infinite capacity (i.e., you should use **My402List** from **warmup assignment #1** to implement them and **not** use arrays).

We will **not** go over the [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 (minor variation is permitted if you document it). Please make sure that your submission conforms to [other general compilation requirements](#) and [README requirements](#).

## Commandline

The command line syntax (also known as "usage information") 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 (according to the description above). 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 *-n* 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 1 (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.

## Running Your Code and Program Output

The emulation should go as follows. At emulation time 0, all 4 threads (arrival thread, token depositing thread, and servers **S1** and **S2** 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 at a time such that the inter-token-arrival time between consecutive tokens is  $1/r$  seconds and would try to deposit one token into the token bucket each time it wakes up. The actual arrival time of the first packet *p1* is denoted as time *T1*, the actual arrival time of the 2nd packet *p2* is denoted as time *T2*, and so on.

As a packet or a token arrives, or as a server becomes free, you need to follow the [operational rules of the token bucket filter](#). Since we have four threads accessing shared data structures, you must use the tricks you learned from Chapter 2 related lectures. Please also check out the [slides for this assignment](#) for the skeleton code for these threads.

You are required to produce a detailed trace for every important event occurred during the emulation and every such event must be **timestamped**. Each line in the trace must correspond to one of the following situations:

- If a packet is served by a server (server **S1** is assumed below for illustration), there must be exactly 7 output lines that correspond to this packet. They are:

```
p1 arrives, needs 3 tokens, inter-arrival time = 503.112ms
p1 enters Q1
p1 leaves Q1, time in Q1 = 247.810ms, token bucket now has 0 token
p1 enters Q2
p1 leaves Q2, time in Q2 = 0.216ms
p1 begins service at S1, requesting 2850ms of service
p1 departs from S1, service time = 2859.911ms, time in system = 3109.731ms
```

Please note the following:

- The value printed for "inter-arrival time" must equal to the timestamp of the "p1 arrives" event minus the timestamp of the "arrives" event for the previous packet.
- The value printed for "time in Q1" must equal to the timestamp of the "p1 leaves Q1" event minus the timestamp of the "p1 enters Q1" event.
- The value printed for "time in Q2" must equal to the timestamp of the "p1 leaves Q2" event minus the timestamp of the "p1 enters Q2" event.
- The value printed for "requesting ???ms of service" must be the requested service time (which must be an integer) of the corresponding packet.
- The value printed for "service time" must equal to the timestamp of the "p1 departs from S1" event minus the timestamp of the "p1 begins service at S1" event (and it should be larger than the requested service time printed for the "begin service" event).
- The value printed for "time in system" must equal to the timestamp of the "p1 departs from S1" event minus the timestamp of the "p1 arrives" event.

- If a packet is dropped, you must print:

```
p1 arrives, needs 3 tokens, inter-arrival time = 503.112ms, dropped
```

Please note that the value printed for "inter-arrival time" must equal to the timestamp of the "p1 arrives" event minus the timestamp of the "arrives" event for the previous packet.

- If <Ctrl+C> is pressed by the user, you must print the following (and print a '\n' before it to make sure that it lines up with all the other trace printouts):

```
SIGINT caught, no new packets or tokens will be allowed
```

Please understand that in order for the above to get printed correctly in a trace printout, using a signal handler to catch signals may not work. You are strongly advised to use a separate SIGINT-catching thread and uses `sigwait()`.

- If a packet is removed when it's in Q# (Q1 or Q2) because <Ctrl+C> is pressed by the user, you must print:

```
p1 removed from Q#
```

- If a token is accepted, you must print:

```
token t1 arrives, token bucket now has 1 token
```

- If a token is dropped, you must print:

```
token t1 arrives, dropped
```

- When you are ready to start your emulation, you must print:

```
emulation begins
```

- When you are ready to end your emulation, you must print:

```
emulation ends
```

All the numeric values above are made up. You must replace them with the actual packet number, actual number of tokens required, actual server number, measured inter-arrival time, measured time spent in Q1, actual number of tokens left behind when a packet is moved into Q2, measured time spent in Q2, measured service time, and measured time in the system.

The output **format** of your program **must** satisfy the following requirements.

- You must first print all the emulation parameters. 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 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 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 which server the packet entered, the time the 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, for example, "no packet was served"). Please note that your program output must **never** contain any "NaN" (which means "not-a-number").

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:
  number to arrive = 20
  lambda = 2          (print this line only if -t is not specified)
  mu = 0.35          (print this line only if -t is not specified)
  r = 4
```

```

B = 10
P = 3                (print this line only if -t is not specified)
tsfile = FILENAME    (print this line only if -t is specified)

00000000.000ms: emulation begins
00000250.726ms: token t1 arrives, token bucket now has 1 token
00000501.031ms: 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 leaves Q2, time in Q2 = 0.216ms
00000752.982ms: p1 begins service at S1, requesting 2850ms of service
00001004.271ms: p2 arrives, needs 3 tokens, inter-arrival time = 501.159ms
00001004.526ms: p2 enters Q1
00001005.615ms: token t4 arrives, token bucket now has 1 token
00001256.259ms: 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
00001508.874ms: p2 leaves Q2, time in Q2 = 0.113ms
00001508.895ms: p2 begins service at S2, requesting 1900ms of service
...
00003427.557ms: p2 departs from S2, service time = 1918.662ms, time in system = 2423.286ms
00003612.843ms: p1 departs from S1, service time = 2859.861ms, time in system = 3109.731ms
...
????????.???ms: p20 departs from S?, service time = ????.???ms, time in system = ????.???ms
????????.???ms: emulation ends

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 S1 = <real-value>
average number of packets at S2 = <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>

```

In the Emulation Parameters section, please print the emulation parameters specified by the user or the default values mentioned above. Please do **not** print the "adjusted" values because certain parameters are too small. (For example, if lambda is 0.01, you must print 0.01 and not 0.1.)

After Emulation Parameters section comes the Event Trace section. The first column there contains **timestamps** and they correspond to event times, measured relative to the start of the emulation. Every emulation event must be timestamped. You need to figure out how to make sure that the timestamp values look reasonable (e.g., never decrease in value). Please use 8 digits (with leading zeroes) 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.

In the Statistics section, the **average number of packets** at a facility can be obtained by adding up all the time spent at that facility (for all relevant 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 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.

All real values in the Emulation Parameters and Statistics sections 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 note that the **timestamps must have microsecond resolution**.

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  $X$ . Please note that we must use the "[population variance](#)" (and not a "sample variance") in our calculation since we have all the data points. Let  $E(X)$  denote the average of  $X$ , you can write:



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

When you are keep statistics, you should [keep a running average](#).

Please note that it's **very important** that the event time in the printout is **monotonically increasing** (as shown in the [sample printout below](#)). This can be difficult to achieve when we have multiple threads running in parallel. But since we are using only one mutex, you can use the following simple (although not super-efficient) trick. When you are getting the time for an event, you must have the mutex locked, and you must not release the mutex until you have printed the line of printout that corresponds to that event, i.e., reading the clock and printing out the event is done in **one atomic operation**.

If the user presses <Ctrl+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 transmitting/servicing the current packet, and output statistics. (Please note that it may not be possible to remove all packets in **Q1** at the instance of signal delivery. The idea here is that once signal delivery has occurred, the only packet you should serve are the ones currently being transmitted/serviced. All other packets should be removed from the system.)

You can divide the packets into 3 categories.

1. **Completed packets:** these are the packets that made it all the way to the server and completed service at the server.
2. **Dropped packets:** these are the packets arrived into the system but never made it even to Q1 because it needs too many tokens.
3. **Removed packets:** these are the packets that got into Q1 to begin with but never made it to the server.

All packets should participate in the calculation of the average packet inter-arrival time and packet drop probability statistics. Only completed packets should participate in the calculation of the average packet service time statistics. Only completed packets should participate in the calculation of the average number of packets in Q1/Q2/S1/S2 and time spent in system statistics.

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 arrive 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 a positive integer which corresponds to the value of **n**. Line **k** of the file contains the inter-arrival time in milliseconds (a positive integer), the number of tokens required (a positive integer), and service time in milliseconds (a positive integer) for packet **k-1**. The 3 fields are separated by space or tab characters (or any combination of any number of these characters). There must be no leading or trailing space or tab characters in a line. If a line is longer than 1,024 characters (including the '\n' at the end of a line), it is considered an error. [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.

In the above example, you should treat these numeric values as "**targets**" or your emulation. In your trace output, you need to print **what you measured** (i.e., by reading the clock). It should be very unlikely that a **measured inter-arrival time** or a **measured service time** has **exactly the same value** as its corresponding target value. For example, the inter-arrival time of packet 3 is suppose to be 972 milliseconds. If the reported actual inter-arrival time between packets 2 and 3 is exactly 972.000 milliseconds, you should look for bugs in your code! Actually, you should probably get a different value every time your rerun your emulation.

This file is expected to be error-free. (This means that if you detect a real error in a **tsfile**, you must simply print an error message and call `exit()` immediately. You **MUST NOT** print statistics or attempt to recover from error in this case.)

You are expected to create your own **tsfile** to test your program. Make sure you know how to create test cases where you know for sure that packets will be wait in Q1, in Q2, or both. You should be able to look at your **tsfile** and predict what will happen in the trace and verify that your program printout is consistent with your prediction.

### Minimum Emulation Time

If you have the fastest machine in the universe that there is no overhead anywhere (i.e., bookkeeping time is zero everywhere, takes zero time to execute any code, etc.) and it's running a real-time OS that always sleeps exactly the amount of time you ask it to sleep, what would be the **minimum simulation time** when you run `warmup2`? Of course, this depends on the parameters of your simulation. Let's take the [sample tsfile shown above](#) and think about when each packet will leave the simulation if we simply run:

./warmup2 -t [tsfile.txt](#)

1. If there is no overhead anywhere, packet p1 would arrive at exactly 2716ms into the simulation. At that time, the token bucket should have more than enough tokens for p1 and p1 would start transmitting immediately. Since the transmission time of p1 is 9253ms, p1 should finish transmission at time 11969ms.
2. Packet p2 would arrive at exactly 7721ms after the arrival time of packet p1. This means that packet p2 would arrive at time 10437ms. At that time, the token bucket should have more than enough tokens for p2 and p2 would start transmitting immediately. Since the transmission time of p2 is 15149ms, p2 should finish transmission at time 25586ms.
3. Packet p3 would arrive at exactly 972ms after the arrival time of packet p2. This means that packet p3 would arrive at time 11409ms. At that time, the token bucket should have more than enough tokens for p3. But, both servers are busy. Therefore, p3 must wait in Q2. The server that transmitted p1 would finish first at time 11969ms and it would start transmitting p3 as soon as it becomes available. Since the transmission time of p3 is 2614ms, p3 should finish transmission at time 14583ms.

From the above analysis, simulation will end when p2 is transmitted at time 25586ms. By doing analysis like this, you can figure out the minimum simulation time of your program. If your program runs faster than that, you would know for sure that you have a bug in your code! (Of course, if the number of packets is large in an input file, it may not be feasible to do this type of analysis by hand.)

### Grading Guidelines

The [grading guidelines](#) and grading data (in [w2data.tar.gz](#)) have been made available. Please run the scripts in the guidelines on a standard 32-bit Ubuntu 16.04 system. 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.)

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** when grading your submission. (We may 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.

For your convenience, a copy of the grading scripts are made available here:

[section-A.csh](#)  
[section-B.sh](#)  
[section-A-all.sh](#)  
[section-B-all.sh](#)

(Technically speaking, the scripts above are not "grading scripts" since they are just scripts provided for your convenience to save you some typing. The grader will not run these scripts when grading.)

After you have download the above shell scripts, please put them in the same directory as warmup2 and run the following command:

```
chmod 755 section-*.sh section-*.csh
```

Please also follow the beginning part of the [grading guidelines](#) to unpack the grading data file (i.e., [w2data.tar.gz](#)) so that the script can work properly. By the way, please do not run these grading scripts in a [shared folder](#). For unknown reasons, running the grading scripts from within a shared folder may not work correctly!

A Perl script, "[analyze-trace.txt](#)", is made available to help you to debug your program printout. (I have to name it as if it's a text file. Otherwise, my web server would try to execute the Perl script.) Please read the comment at the top of the code to see how to use it. This code only works if your printout is in the right format and each regular packet has 7 lines of printout (with their timestamps in chronological order) and each dropped packet has 1 line of printout. If your printout has missing lines in the printout or if the lines are in the wrong order, you should fix your code and rerun this script!

For this assignments, please always use `pthread_cond_broadcast()` to wake up server threads. Please do not use `pthread_cond_signal()` anywhere in your code. (Yes, this is not the **most** efficient way. But since the grader must follow the grading guidelines when grading, this would most likely get you the most number of points.)

In section (A) of the [grading guidelines](#), each test has a minimum emulation time. The numbers were obtained using the [Minimum Emulation Time analysis mentioned above](#). If the running time of your code is too fast or too slow, it means that you have pretty serious bugs in your code and you need to get them fixed or you will end up losing a lot of points.

### Miscellaneous Requirements & Hints

- Please read the [general programming FAQ](#) if you need a refresher on file I/O and bit/byte manipulations in C.
- You must **NOT use any external code segments** to implement this assignment. You must implement all these functionalities from scratch.
- You must **NOT use semaphores** to implement this assignment. You must implement thread synchronization using `pthread_mutex` and `condition variable`. If you use something like a semaphore, you will lose a lot of points!

- Please do not use an array to store all the packets. If you do that, you may end up losing a lot of points because there will be cases your program will not be able to handle. Please design your program so that it can handle billions of packets. If you don't know how to do this, you should start a discussion in the class Google Group.
- It's probably not a good idea to keep timestamps in double because you need to worry about round-off errors when you add or subtract timestamps. I would **strongly recommend** that you keep timestamps in `struct timeval`, which is the data structure used by `gettimeofday()` to represent time. This data structure contains two integers (`tv_sec` is the number of seconds since 1/1/1970 and `tv_usec` is the number of microseconds in the current second) so you don't have to worry about round-off errors. But you need to worry about carry when you add timestamps and borrow when you subtract timestamps. To add timestamps, you can use `timeradd()`, and to subtract timestamps, you can use `timersub()`. Please do `"man timeradd"` to see how to use these functions. You may have to write additional utility functions to do things like displaying timestamps (for the first column of your trace printout), displaying intervals (for other types of trace printout), etc.
- 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**.
- I do **not** recommend using a signal handler to catch SIGINT. You are strongly advised to use a separate SIGINT-catching thread and use `sigwait()`.
- Removing packets are part of the emulation. Therefore, removing of each packet must be timestamped and you must not terminate your emulation before you have removed all the packets.
- If `<Ctrl+C>` is pressed and you print the "SIGINT caught" message in the printout, you must print a timestamp for this event.
- Your code must not "busy-wait"! "Busy-waiting" means that you have code that's staying in a tight loop to wait for some condition to become true. Such code is very unfriendly to the environment you are running your program in. Therefore, you must not do busy-waiting. If you find a piece of busy-waiting code, just insert `"usleep(100000)"` inside the loop to sleep for 0.1 second before checking the condition again. Since it's so easy to not do busy-waiting, if the grader sees that your code is doing busy-waiting, a lot of points will be deducted. (Please understand that this is just a quick and dirty fix! The correct way to wait for some condition to become true is to sleep in a CV queue as discussed in lectures.)

If your code is not doing anything useful (i.e., just waiting for something) and you run "top" from the commandline on your 32-bit Ubuntu 16.04 system and you see that your program is taking up one of the top spots in CPU percentages and showing something like more than 0.5%, there is a good chance that you have busy-waiting code. In this case, run your code under `gdb` and run "top" in another terminal. When your code is not doing anything useful and your process starts to show up in "top", press `<Ctrl+C>` in `gdb`. Hopefully, your program will break inside your busy-waiting loop (use the `where` command to see where you are inside `gdb`!).

- Please understand the meaning of **inter-arrival time**. It means **time between consecutive packets**. Let's say the inter-arrival time is 1 second and we are running in the deterministic mode. Does it mean that packet 1 will arrive at 1 second after the start of emulation, packet 2 will arrive at 2 second after the start of emulation, packet 3 will arrive at 3 second after the start of emulation, etc.? No. You need to **schedule** packet 1 to arrive at 1 second after the start of emulation. But packet 1 will most likely **not** arrive until a little bit later than 1 second after the start of emulation. You record the **actual time of arrival** of packet 1 and you add 1 second to it and that would give you the **expected arrival time** of packet 2. Then you can **schedule** packet 2 to arrive at its **expected arrival time**.

Here are some additional hints:

- For this assignment, you are implementing a **time-driven** emulation (and *not* an event-driven simulation such as the ns-2 in networking). 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 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 seems 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 sleeping for the server thread but only applies to the packet arrival and token depositing threads.)
- Continue with the above example, if it took more than 634 milliseconds to do bookkeeping and to enqueue the packet to the queueing system, what should you do? Since you cannot sleep for a negative number of microseconds, you should skip sleeping. This is "**best-effort**" emulation. You are **not required** to hit the target time, you just need to try your best to match inter-arrival times. When you cannot, you should still try your best (i.e., 0 is the closest number to a negative number).



## Submission

All assignments are to be submitted electronically (including the required "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. The command you can use to create a gzipped tarfile is:

```
tar cvzf warmup2.tar.gz MYLISTOFFILES
ls -l warmup2.tar.gz
```

Where MYLISTOFFILES is a 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** for each binary file included in your submission (e.g., warmup2, .o, .gch, core, etc.). The last command shows you how big the created "warmup2.tar.gz" file is. If "warmup2.tar.gz" is larger than 1MB in size, the submission server will not accept it.

Please note that the 2nd commandline argument of the tar command above is the **output** filename of the tar command. So, if you omit warmup2.tar.gz above, you may accidentally replace one of your files with the output of the tar command and there is no way to recover the lost file (unless you have made a backup copy). So, please make sure that the first commandline argument is **cvzf** and the 2nd commandline argument is **warmup2.tar.gz**.

A [w2-README.txt template file is provided here](#). You must save it as your w2-README.txt file and follow the instructions in it to fill it out with appropriate information and include it in your submission. You must not delete a single line from w2-README.txt. Please make sure that you [satisfy all the README requirements](#).

Here is an example commands for creating your warmup2.tar.gz file:

```
tar cvzf warmup2.tar.gz Makefile *.c *.h w2-README.txt
```

If you use an IDE, you need to modify the commands above so that you include **ALL** the necessary source files and subdirectories and make sure you have not forgotten to include a necessary file in order for the grader to compile your code on a "clean" system.

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.

To check the content of warmup2.tar.gz, you can use the following command:

```
tar tvzf warmup2.tar.gz
```

Please read the output of the above command carefully to see what files were included in warmup2.tar.gz and what are their file sizes and make sure that they make sense.

Please enter your **USC e-mail address** and your **submission PIN** below. Then click on the **Browse** button and locate and select your submission file (i.e., warmup2.tar.gz). Then click on the **Upload** button to submit your warmup2.tar.gz. (Be careful what you click! Do **NOT** submit the wrong file!) If you see an error message, please read the dialogbox carefully and fix what needs to be fixed and repeat the procedure. If you don't know your submission PIN, please [visit this web site to have your PIN e-mailed to your USC e-mail address](#).

When this web page was **last loaded**, the time at the submission server ([merlot.usc.edu](#)) was **30Aug2022-16:05:17**. **Reload** this web page to see the **current time** on [merlot.usc.edu](#).

USC E-mail:	<input type="text"/>	@usc.edu
Submission PIN:	<input type="text"/>	
Event ID (read-only):	merlot.usc.edu_80_1557931083_237	
Submission File Full Path:	<input type="button" value="Choose File"/>	No file chosen
	<input type="button" value="Upload"/>	

If the command is executed successfully and if everything checks out, a **ticket** will be issued to you to let you know "what" and "when" your submission made it to the Bistro server. The next web page you see would display such a ticket and the ticket should look like [the sample shown in the submission web page](#) (of course, the actual text would be different, but the format should be similar). Make sure you follow the [Verify Your Ticket](#) instructions to verify the SHA1 hash of your submission to make sure what you did not accidentally submit the wrong file. Also, an e-mail (showing the ticket) will be sent to your USC e-mail address. Please read the ticket carefully to know exactly "what" and "when" your submission made it to the Bistro server. 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 every you have submitted is everything you wanted us to grade. If you don't [verify your submission](#) and you ended up submit the wrong files, please understand that due to our fairness policy, there's absolutely nothing we can do.

Finally, please be familiar with the [Electronic Submission Guidelines](#) and information on the [bsubmit web page](#).

---

[Last updated Fri Aug 19 2022] [Please see [copyright](#)  
regarding copying.]

[ Home | Description | Lectures | Videos | Discussions |  
Projects | Forum ]