CS 324, Winter 2019

Proxy Lab: Writing a Pre-threaded Web Proxy with a Cache

Due: Friday, Mar. 29, 11:59 PM

1 Introduction

In this lab, you will:

- Change the proxy server you wrote for the last lab to use a pool of threads instead of launching a new thread for each request. This is described in Section 12.5.5 of your textbook.
- Add a cache to your proxy that stores recently-used Web objects in memory.
- Add logging to a file.

2 Logistics

This is an individual project.

3 Handout instructions

You will work off the same set of files as you did for the previous lab.

4 Threadpool

Launching a thread for each request is costly and could use all your resources. A better approach is to launch several threads (a *threadpool*) that wait for tasks such as handling a client.

In this lab, you will implement two sets of threads: one set to handle client connections and the other to handle logging. Both sets of threads will operate in producer-consumer fashion. They are explained in more detail in subsequent sections.

4.1 Handling Clients

When your proxy starts up, you will start a set of threads whose job is to handle clients requests. The client sockets handled by these threads are handed to them by the main thread. Thus, the main thread becomes the producer, and the client handler threads are the consumers.

The main thread adds the socket corresponding to each client connection to a shared queue, and the client handling threads retrieve these sockets from the shared queue and perform the proxy functions. Both producer and consumer wait on semaphores to wait for slots and items, respectively, as well as to protect shared access to the queue. In this way, you limit the number of resources your server uses.

4.2 Logging

In addition to the client handling threads that you create on startup, your proxy should start up a single logging thread. When this thread starts, it will open a file for writing, after which it will simply log messages from the client handling threads and write those log messages to the file it opened. Each log message simply needs the URL requested, one per line, but you can make it fancy with a timestamp and the client IP address, if you so choose.

As with the producer-consumer model used to hand off client connections from the main thread to the pool of threads handling those connections. you will need a separate shared queue for passing log messsages from client handling threads to the logging thread. In this case, the client handling threads become the producers, and the loging thread is the consumer!

4.3 Other Notes

Note: Access to both queues will need to be protected with semaphores so the operations are thread safe. The queues should use producer/consumer style signalling with the semaphores.

5 Caching

HTTP actually defines a fairly complex model by which web servers can give instructions as to how the objects they serve should be cached and clients can specify how caches should be used on their behalf. However, your proxy will adopt a simplified approach.

When your proxy receives a web object from a server, it should cache it in memory as it transmits the object to the client. If another client requests the same object from the same server, your proxy need not reconnect to the server; it can simply resend the cached object.

Obviously, if your proxy were to cache every object that is ever requested, it would require an unlimited amount of memory. Moreover, because some web objects are larger than others, it might be the case that one giant object will consume the entire cache, preventing other objects from being cached at all. To avoid those problems, your proxy should have both a maximum cache size and a maximum cache object size.

5.1 Maximum cache size

The entirety of your proxy's cache should have the following maximum size:

```
MAX_CACHE_SIZE = 1 MiB
```

When calculating the size of its cache, your proxy must only count bytes used to store the actual web objects; any extraneous bytes, including metadata, should be ignored.

5.2 Maximum object size

Your proxy should only cache web objects that do not exceed the following maximum size:

```
MAX_OBJECT_SIZE = 100 KiB
```

For your convenience, both size limits are provided as macros in `proxy.c`.

The easiest way to implement a correct cache is to allocate a buffer for each active connection and accumulate data as it is received from the server. If the size of the buffer ever exceeds the maximum object size, the buffer can be discarded. If the entirety of the web server's response is read before the maximum object size is exceeded, then the object can be cached. Using this scheme, the maximum amount of data your proxy will ever use for web objects is the following, where `T` is the maximum number of active connections:

```
MAX_CACHE_SIZE + T * MAX_OBJECT_SIZE
```

5.3 Eviction policy

No eviction policy required for this lab. If there is not room in the cache, then just don't cache the object.

5.4 Synchronization

Accesses to the cache must be thread-safe, and ensuring that cache access is free of race conditions will likely be the more interesting aspect of this part of the lab. As a matter of fact, there is a special requirement that multiple threads must be able to simultaneously read from the cache. Of course, only one thread should be permitted to write to the cache at a time, but that restriction must not exist for readers.

As such, protecting accesses to the cache with one large exclusive lock is not an acceptable solution. You may want to explore options such as partitioning the cache, using Pthreads readers-writers locks, or using semaphores to implement your own readers-writers solution. In either case, the fact that you don't have to implement a strictly LRU eviction policy will give you some flexibility in supporting multiple readers.

6 Autograding

Your handout materials include an autograder, called `driver.sh`, which will evaluate your proxy server. Use the following command line to run the driver for this lab:

```
linux> ./driver.sh threadpool 15 35 10 2 35
```

This will provide you a grade and feedback for your assignment. Note that maximum possible points that the script will output is 97. The remaining three points are for no warnings with compilation.

7 Resources and Hints

You are welcome to use the shared buffer code provided in your concurrency homework (i.e., `buf.c`). Of course, you will need to modify it to fit your needs, rather than using it as is.

You can also use the Readers-Writers code from the textbook and the slides to implement the code for accessing and updating your cache.

While a real proxy implementation will need to be fast and efficient, in this lab, we are simply looking for correctness. That being said, here are some things to keep in mind:

- You can simply read a server response until you have it all, then store it all in the cache and send it to the client, rather than sending to the client and/or caching it as you read it.
- You can store the response headers and content (i.e., response body) in your cache, if you'd like. While that's not how it's done in a real cache, this will simplify the job for you.
- You can simply store a complete URL with each cache entry and then iterate through each cache entry, checking the requested URL against each stored URL to find a match. While hashing would be much faster, we're not concerned about this aspect of performance.

8 Grading

The following is the point breakdown:

- - 15 points for basic proxy operation
- - 35 points for handling concurrent requests (with a threadpool)
- - 35 points for implementing a working cache
- - 10 points for logging URLs to a file
- - 3 compiles without any warnings
- - 2 no memory leaks

The maxmimum number of points is 100.

9 Handin instructions

The provided Makefile includes functionality to build your final handin file. Issue the following command from your working directory:

```
linux> make handin
```

The output is the file `../proxylab-handin.tar`, which you can then handin.

To submit your lab, please upload your `proxylab-handin.tar` file to the appropriate assignment page on Learning Suite.