Instructor: Ashvin Goel
Course Number: ECE344

Home
Lecture Notes
Lab Assignments
Discussion (piazza)
Grades (UofT portal)

# Operating Systems

**ECE344, FALL 2016**
**UNIVERSITY OF TORONTO**

## LAB 5: A MULTI-THREADED WEB SERVER

**Due Date: Nov 21, 2016, 5 pm**

In this lab, you will be developing a caching web server, and evaluating its performance. You will start with the multi-threaded web server you have built in Lab 4, and add file caching to your web server, to make it more efficient.

## SETUP

You will be doing this lab within the `webserver` directory that you created in Lab 4.

Make sure to commit all your previous changes (or discard any uncommitted changes) in your local repository, and then run the following commands to get started with this lab.

```
cd ~/ece344
git tag Lab5-start
cd webserver
make
```

## CACHING WEB SERVER

The multi-threaded web server you have built in Lab 4 serves web requests efficiently by using multiple threads. Using multiple threads has two benefits. First, it allows using the multiple cores of your machine, so that requests can be served concurrently. Second, it allows processing a request, while another request is fetching a file from disk.

However, your multi-threaded server processes each file request by fetching the file from disk. This slows down request processing dramatically, because disk accesses can be very slow (roughly 10 ms per file access). Furthermore, this slows the overall throughput of the web server because the disk speed becomes the bottleneck for your web server.

In this lab, you will address this limitation of your multi-threaded web server by caching files that have been recently requested in memory. If these files are requested again, then they can be served from memory instead of needing to fetch the file from disk. This is specially beneficial when some files are requested much more often than others.

A file consists of a sequence of blocks of a fixed size (e.g., 4KB). File caching can be implemented in at least two different ways. Files can be cached at a *block granularity* (some file blocks may be cached, while others may not be cached), or files can be cached at whole *file granularity* (a file is cached in its entirety or not cached at all). The benefit of block granularity caching is that if only parts of a file are accessed, then the entire file does not have to be cached. Also, if a file doesn't fit in available memory, then the entire file cannot be cached. Finally, memory management is easier because blocks are of fixed size. Hence, operating systems typically implement block granularity caching.

However, in this project, you will be using file granularity caching in the web server, for three reasons. First, the web server will be reading entire files. Second, we will ensure that these files will fit in memory. Third, we can use our favorite `malloc()` and `free` library functions to allocate memory for caching whole files, without worrying about fragmentation caused by allocating variable-sized chunks of memory.

Alright, at this point, you might be thinking that if the operating system is already caching files (at block granularity), then is there any benefit to caching files in the web server? Generally, it is not a good idea to cache data twice in the same level of the memory hierarchy because it wastes memory. To avoid double caching, the code we have provided disables caching of files in the operating system (what line of code does it?). Hence, your web server cache should improve your server performance.

A benefit of caching in the web server, compared to file caching in the operating system, is that your server program will have complete control over which files to cache and which files to evict (i.e., the cache replacement policy). For example, you may choose to avoid caching very large files, or files that are unlikely to be requested often. Similarly, you may choose to evict larger files before smaller files. The web server can make these decisions better than the operating system, because it has more information about its requirements.

In your cache implementation, you should use a hash table to lookup cached files. The key (or tag) for the cache lookup should be the file name, and the data should be the file data that is fetched from disk. Luckily, you have already implemented a string-based hash table in Lab 1, so you should be able to reuse that code for your cache.

But all is not done yet.

Your cache must only be allowed to grow to a maximum limit. Recall that the web server that we have provided takes a `max_cache_size` argument. This argument should be the maximum amount of bytes that are used by all the files in your file cache (i.e., the sum of the file sizes of all the cached files must be less than or equal to `max_cache_size`). For this lab, you should run the web server with a positive value for this argument (e.g., 100000). Increasing this value should improve the performance of your web server.

How can you ensure that your cache doesn't grow beyond the `max_cache_size` limit? When you try inserting a newly fetched file in your cache, you may need to evict other files, if this limit would be reached by caching the new file.

We suggest implementing these three functions for this lab: `cache_lookup(file)`, `cache_insert(file)`, and `cache_evict(amount_to_evict)`. You may add any other parameters to these functions.

These functions will be accessed by multiple threads, so you will need to use locks to ensure mutual exclusion. A simple locking scheme is to use a single lock that protects the entire cache. If you use this scheme, your code **must not** hold this "global" lock while reading a file from disk or sending it to the client. These I/O operations take a long time, and holding a global lock while performing I/O will serialize all threads, negating any benefits of the multi-threaded web server over the basic web server.

You will need to implement an eviction algorithm. A simple (and effective) eviction algorithm is the least-recently used (LRU) algorithm, which discards the least-recently used file first. To implement LRU, you will need to keep an LRU list of the cached files, and update this list on a cache lookup or insert operation. However, you are welcome to use any eviction algorithm of your choice, including evicting larger files first, etc., as discussed earlier.

## SOLUTION REQUIREMENTS

As mentioned above, your code must not hold any global lock while performing file and network I/O operations.

Your cache **must not** store multiple copies of a file. How might this happen? Consider what happens if while reading a file from disk to serve a request, another request occurs for the same file, and the thread serving this request also reads the file from disk, and populates the cache. Now the first request finishes reading the file, and populates the cache with another copy of the file. **If your implementation performs any busy-waiting (or spin-waiting) to handle this case, you will be heavily penalized.**

## HINTS AND ADVICE

This project does not require writing a large number of lines of code. It does require you to think carefully about the code you write. Before you dive into writing code, it will pay to spend time planning and understanding the code you are going to write. If you think the problem through from beginning to end, this project will not be too hard. If you try to hack your way out of trouble, you will spend many frustrating nights in the lab. This project's main difficulty is in conceptualizing the solution. Once you overcome that hurdle, you will be surprised at the simplicity of the implementation!

As a start, here are some questions you should answer before you write code.

- How will you ensure that files that are in use are not evicted? For example, the server might be sending a cached file to the client. Evicting this file will deallocate the in-memory copy of the file, possibly sending garbage to the client, or crashing your server.

- How will you ensure that files are not multiply cached? One option might be to allow threads to read the same file concurrently, if the file is not cached currently. On returning from the file read, a thread could check whether the file has already been cached (it lost the race). If so, it can avoid caching its file copy (i.e., free the buffer containing the file contents). Another better option might be to synchronize requests for reading the same file from disk. In either case, make sure you think about locking and synchronization carefully.
- Are there any cases when enough files cannot be evicted, when the `evict()` function is called? What should happen in this case?
- What happens if the file size is greater than the cache size?
- What data structure will you use to implement your cache eviction algorithm? Will it require its own lock and synchronization, or can you reuse any other locks?

You are encouraged to reuse *your own* code that you might have developed in the previous labs or in previous courses to handle things such as queues, hashing, etc.

## TESTING YOUR CODE

To help you test your code better, we have provided several scripts, as described below.

**run-one-experiment:**

This script runs the `server` program, and then it runs the `client` program 10 times, and provides the average run time of the client. All the client run times are recorded in the `run.out` file. Read the beginning of this file to see how it should be invoked. For this lab, you should use this script to run the server with varying cache sizes. This script can take up to 1-2 minutes to run.

**run-cache-experiment:**

This script runs the `run-one-experiment` script while varying the caching-related server parameters. Read the beginning of this file to see how it should be invoked. This script will generate one output file called `plot-cachesize.out`. This script can take 5-10 minutes to run, so use it when you are close to finishing the lab.

**plot-cache-experiment:**

This script plots the output file generated by `run-cache-experiment`. The plot is available in `plot-cachesize.pdf`.

You can test your entire code by using our auto-tester program at any time by following the testing instructions. For this assignment, the auto-tester program simply runs the `run-experiment` and the `plot-experiment` scripts.

## USING GIT

You should only modify the following files in this lab.

```
  server_thread.c
```

You can find the files you have modified by running the `git status` command.

You can commit your modified files to your local repository as follows:

```
  git add server_thread.c
  git commit -m "Committing changes for Lab 5"
```

We suggest committing your changes frequently by rerunning the commands above (with different meaningful messages to the commit command), so that you can go back to see the changes you have made over time, if needed.

Once you have tested your code, **and committed it** (check that by running `git status`), you can tag the assignment as done.

```
git tag Lab5-end
```

This tag names the last commit, and you can see that using the `git log` or the `git show` commands.

If you want to see all the changes you have made in this lab, you can run the following `git diff` command.

```
git diff Lab5-start Lab5-end
```

More information for using the various git commands is available in the Lab 1 instructions.

## CODE SUBMISSION

Make sure to add the `Lab5-end` tag to your local repository as described above. Then run the following command to update your remote repository:

```
git push
git push --tags
```

For more details regarding code submission, please follow the lab submission instructions.

Please also make sure to test whether your submission succeeded by simulating our automated marker.