CSCI3120
Assignment 4*

Instructor: Alex Brodsky

Due: 12:00pm (noon), Friday, July 29, 2016

The purpose of this assignment is to get you to do more thread programming and process synchronization, and also reinforce our discussion of caching from class. In this assignment you build upon Assignment 3, to create a Caching Multithreaded Scheduling Web Server. You will:

1. Implement cache in your Assignment 3 solution (or the one provided), and
2. Create a test suite for the caching multithreaded scheduling web server.

You have a choice: You can either use your code from Assignment 3 as the starting point, or you can use the provided solution to Assignment 3. Regardless, you need to ensure that you have `assn4` checked out in your SVN repository. This directory will contain the solution to Assignment 3. Please follow the guidelines in Section 5.1.

1 Background

Our goal is to make our web server faster. In many instances, servers serve the same files repeatedly to many clients. For example, the main page of website gets served very often to almost every client that visits the website. Thus, it makes sense to cache these files, obviating the need to read them from disk every time they are requested. This not only saves quite a few system calls, but also the time it takes to load the file from disk. There are two issues that your cache implementation will need to address: i) a file may have several different names, and ii) what happens when the cache fills up.

Recall from our discussion in class, a file may actually have several different names if it has multiple links to it. There is no point in storing multiple copies of the same file in the cache, so your server's cache implementation will need to ensure that only one copy of a file is stored in the cache at any time. Furthermore, if a request for file F arrives while file F is being loaded into the cache, the request should block until F is loaded into the cache. I.e., at no time should a file be loaded multiple times into the cache.

Inevitably, the server's cache will fill. At this point, the next file to be loaded will require the eviction of another file. The cache should use the Least Recently Used (LRU) scheme for deciding which file to evict. However, any file that is currently requested should never be evicted. Furthermore, If a file cannot be loaded into the cache, because it is either too big, or not enough files can be evicted to make room, the file will need to be read from disk instead.

To implement the functionality described above your web server will likely need to use a file control block for each file stored in the cache, and keep a list of files currently in the cache. Recall that the i-node number of the file is the file's unique identifier, and can be used to distinguish between two different files and two different names of the same file. Lastly, since your web-server is multithreaded, your cache will also need to be thread-safe.

*Revised: July 11, 2016

2 Task 1: The Caching Multithreaded Scheduling Web Server

Your task is to implement the caching multithreaded scheduling web server (sws). This web server should have the same functionality as the multithreaded scheduling web server (Assignment 3) and have the additional functionality:

1. The program should take a fourth argument, which is a positive integer denoting the size of the cache. For example, this invocation binds to port 12345 on start up, uses the SJF scheduler, creates 64 worker threads, and creates a 1MB cache.

   ```
   ./sws 12345 SJF 64 1048576
   ```

2. When processing a request, instead of using the `fopen()` / `fclose()` / `fread()` API to load the file, the thread should use the cache's API.
3. The cache should provide the following API:

   `void cache_init( int size )`
   Allocate a cache of the specified size. **The cache should allocate a piece of memory of the specified size, and all file data must be stored in that one piece of memory.** Any data structures to manage the loaded files may be allocated separately.

   `int cache_open( char *file )`
   If the file is not in the cache, load it into the cache. Return a "cached file descriptor" (CFD), which refers to the file in the cache. Note, each worker thread should get its own CFD when opening a file. This function returns -1 if the file cannot be accessed.

   If a thread requests a file that is being loaded into the cache, the thread should block until the file is loaded.

   If the file cannot be loaded because the file is too big, a CFD should still be returned. The file should be opened and the cache should read the file from disk when it is needed, instead of storing it in memory.

   After a file is loaded into the cache, output to `stdout`

   ```
   File of size <size> cached.
   ```

   and after a file is evicted from the cache, output to `stdout`

   ```
   File of size <size> evicted.
   ```

   where `<size>` is the size of the file.

   `int cache_send( int cfd, int client, int n )`
   The next n bytes of the file referred to by the *cfd* should be sent to the client by performing a write with the (*client*) file descriptor. The value n is the maximum number of bytes to send. If the number of bytes remaining is $r < n$, then r bytes should be sent. The function returns the number of bytes sent.

   If the file referred to by the *cfd* is too big to fit into the cache, it should be read from disk instead of from the cache.

   `int cache_filesize( int cfd )`
   Returns the size of the file.

   `int cache_close( int cfd )`
   Closes the cached file descriptor. If the file referred to by the *cfd* is too big to fit into the cache, it should be closed.

Note: after each output to `stdout`, using `printf` or any other other function, please add the call to `fflush( stdout )` to ensure all output is properly displayed.

You can use the provided `makefile` to build your web server simply by running `make`. You can use your solution from Assignment 3, or the provided solution to Assignment 4 as a starting point. Please see Section 3 on how you can test your implementation.

3 Task 2: Test Your Code

You will need to test your scheduling web server implementation. A tool `hydra.py` is provided to help you test your code. This program, written in Python, acts as a multiheaded client that can send concurrent requests to your server. You can schedule when the requests are sent, and time how long the requests take, as well as the response time for each request. You can also compute the average response time for all requests. Please read `hydra.py` to see how to use it. You can use the same procedure as in Assignment 3 to test your code.

Develop 10 test cases to test your caching multithreaded scheduling web server, with the same test cases being used to test all three schedulers. Create three (3) files for each test:

`test.#.in` is the script file for `hydra.py`
`test.#.txt` is a brief description of the test and the expected result
`test.#.$.out` is the expected output of the test

where # represents a number 0 . . . 9, and $ is the scheduler, which is one of `S`, `R`, or `M`. The `.out` file can be created from the `.in` file. See the test scripts for Assignment 2 and Assignment 3 for an example. The tests should test all the functionality of the caching multithreaded scheduling web server. The test description files **must** contain: a title, author (your name) purpose of the test, how the test works, and a description of the expected result.

4 Hints and Suggestions

1. You will need to keep the equivalent of a System-Wide Open File Table to manage the files in the cache, and a separate table of CFDs.
2. You will need to use reference counts to track how many requests are using a particular file in the cache at any time.
3. You will also need to implement LRU in your cache. A linked list is a simple way to do that.
4. Be sure to protect you cache with a mutex!
5. Not a lot of code is required, but identifying what code needs to be changed is challenging.
6. The `pthread_mutex_*` functions are useful for ensuring atomicity.
7. The `pthread_cond_*` functions are useful for suspending/resuming threads holding a mutex.
8. Start early, this assignment cannot be done in one sitting.

5 What to Hand In

You must submit an electronic copy of your assignment. Submissions must be done by 12 noon of the due date. The submission should be done using SVN. See the course web-page (Assignments or Resources) on how to submit using SVN.

5.1 Programming Guidelines

1. Use the file and function names specified in the assignment.
2. Your programs will be compiled and tested on the `bluenose.cs.dal.ca` Unix server. In order to receive credit, your programs must compile and run on this server.
3. All submitted programs must compile. **Programs that do not compile will receive a** 0**.**
4. All submitted programs must be built with the provided makefile. **Programs that do not build with the makefile will receive a** 0**.**
5. The code must be well commented, properly indented, and clearly written. Please see the style guidelines under Assignments or Resources on the course website.

5.2 Required Files

Use SVN to submit the following files.

1. The source code files that implement the multithreaded scheduling web server.
2. The makefile used to build your web server
3. The test files specified in Section 3

6 Grading

	Mark
Simple Web Server	/30
Structure and Design of Web Server	/10
Implementation of Cache	/10
Functionality (automated tests)	/10
Test Suite	/15
Quality (effectiveness of the tests)	/5
Number of tests	/10
Code Clarity	/5
Total	**/50**