CSCI3120
Assignment 3*

Instructor: Alex Brodsky

Due: 12:00pm (noon), Monday, July 11, 2016

The purpose of this assignment is to get you to use threads (pthreads) and to implement some synchronization mechanisms on top of your scheduling server. In this assignment you build upon Assignment 2, to create a Multithreaded Scheduling Web Server. You will:

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

You have a choice: You can either use your code from Assignment 2 as the starting point, or you can use the provided solution to Assignment 2. Regardless, you need to ensure that you have `assn3` checked out in your SVN repository. This directory will contain the solution to Assignment 2. Please follow the guidelines in Section 5.1.

1 Background

Our goal is to make our web server scalable and responsible by using threads to service requests. Currently, if the client does not immediately sends the request, our server blocks, waiting for the request, preventing other requests from being serviced. This denial of service attack is not acceptable in a good web server. We will make our server multithreaded to solve this problem. That is, when `network_open()` returns a valid file descriptor, a thread will be assigned to process the request and submit it to the scheduler, if the request is valid.

Worker threads are assigned from work-queue that is filled by the main thread. Specifically, When the server starts up, the main thread creates a fixed number of worker threads. The main thread is responsible for performing `network_open()`, and when a client connection is returned, the main thread allocates an RCB, stores the client connection in the RCB, and inserts the RCB into the shared work queue. Worker threads remove RCBs from work queue, and start processing the request by reading the request from the client, determining if it's valid, and if it is, submitting the request to the scheduler. If the work queue is empty, worker threads request jobs from the scheduler and perform them using the same rules as in Assignment 2. If both the work queue and the scheduler are empty, the worker threads wait until more requests are added to the work queue.

As you may be already observing, the work queue represents a producer-consumer problem with one producer and many consumers. As well, the scheduler is also a critical section, where only one thread at a time should submit or get the next request. Comparable behaviours occur on multi-CPU systems, where the OS has to scheduler processes on different CPUs.

Note, we do not want our server to swamp the system if too many requests come in. Hence, our server will use a limited number of threads. Furthermore, as mentioned previously, only the main thread accepts new clients. The new clients are passed to the worker threads via the work queue,

*Revised: June 20, 2016

and the clients' requests are scheduled by the selected scheduler. Lastly, for this assignment, you will likely need to add a couple more fields to the RCB.

2 Task 1: The Multithreaded Scheduling Web Server (`sws`)

Your task is to implement the multithreaded scheduling web server (sws). The file `sws.c` should contain the `main()` function. The multithreaded scheduling web server should do the following:

1. The main thread should:
   (a) Take three (3) arguments on the command line:
      i. The first argument is a nonnegative integer denoting the port number to which the web server should bind.
      ii. The second argument is the scheduler that the server should use. Your server should implement (at minimum) the same three schedulers as in Assignment 2:
      SJF : Shortest Job First
      RR : Round Robin
      MLFB : Multilevel queues with Feedback
      iii. The third argument is a positive integer denoting the number of worker threads that should be created.

      For example, this invocation binds to port 12345 on start up, uses the SJF scheduler, and creates 64 worker threads.

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

   (b) Initialize the network module.
   (c) Create and/or initialize a work queue.
   (d) Instantiate and start the specified number of worker threads.
   (e) Enter an infinite main loop that
      i. Checks if there are clients waiting to connect
      ii. If no clients are waiting, wait for clients to connect.
      iii. Connect to each client, allocate an RCB, and insert it into the work queue.
      iv. If a worker thread is waiting for a request, wake it up.

      The server should be able to handle up to 100 concurrent requests. Once if a server is processing 100 requests, it should not call `network_open()` until there are less than 100 concurrent requests.

2. Each worker thread should enter an infinite main loop that does the following:
   (a) If there are requests in the work queue: Dequeue a request and process it in the same manner as in Assignment 2. I..e, read the request from the client, and if the request is valid, admit it to the scheduler.
   (b) Else if, there are no requests in the work queue and there are requests in the scheduler's ready queue to be processed: Select the next request and process it in the same manner as in Assignment 2.
   (c) Else, since there are no requests in the work queue or the scheduler, the thread should suspend itself until it is waked by the main thread.

3. Each client's request is admitted in the same manner as in Assignment 2, except with the following minor change:
   (a) After the request is admitted to the scheduler, output to `stdout`

   ```
   Request for file <file> admitted.
   ```

   where `<file>` is the path of the file being requested.
4. Each client's request is serviced in the same manner as in Assignment 2, except with the following minor change:
   (a) After each *write()* to the client, output to `stdout`

   ```
   Sent <n> bytes of file <file>.
   ```

   where `<n>` is the number of bytes sent and `<file>` is the path of the file being requested.
   (b) After the request to the client is completed, output to `stdout`

   ```
   Request for file <file> completed.
   ```

   where `<file>` is the path of the file being requested.

You can use the provided `makefile` to build your web server simply by running `make`. You can use your solution from Assignment 2, or the provided solution to Assignment 2 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 2 to test your code.

Develop 10 test cases to test your 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. E.g.,

```
./hydra.py < test.1.in  > test.1.M.out
```

The tests should test all the functionality of the 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 can allocate a static request table with 100 RCBs.
2. Not a lot of code is required, but identifying what code needs to be changed is challenging.
3. The `pthread_mutex_*` functions are useful for ensuring atomicity.
4. The `pthread_cond_*` functions are useful for suspending/resuming threads holding a mutex.
5. 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 Work Queue	/5
Protection of Schedulers	/5
Functionality (automated tests)	/10
Test Suite	/15
Quality (effectiveness of the tests)	/5
Number of tests	/10
Code Clarity	/5
Total	**/50**