# Assignment 1

**Problem statement**: Develop a basic multithreaded webserver (where multiple threads process multiple client requests simultaneously) with support for different scheduling policies for the client requests.

To keep this assignment within the scope of OS course, we will be providing a basic webserver code (without multithreading) with this assignment. This basic webserver code is a slightly modified version of Prof. Remzi’s concurrent webserver project (<https://github.com/remzi-arpacidusseau/ostep-projects/tree/master/concurrency-webserver>). You can execute this code and get a feel of how it works (execution instructions are provided in section 3.1). However, you won’t have to modify this code; you’ll have to modify the provided template. This template is specifically designed to make sure that you’ll only need to work on the OS concepts rather than worrying about the socket programming or managing the project with multiple files. **You’ll have to modify only a single file from the template (request.c) and submit only that one file in gradescope**. Do remember to strictly follow the instruction given in section 3 (implementation details) while modifying the code. Please go through below sections for the assignment details.

1. **Basic webserver code (provided with this assignment)**

The provided code has a web server and a web client (similar to a web browser). The web server and the web client interact using HTTP (Hypertext Transfer Protocol). Web client opens a connection to a web server and requests some content with HTTP. The web server responds with the requested content and closes the connection. The web client then reads the content and displays it on the screen. You can even use any web browser instead of the provided web client and it shall work fine for http requests.

The content supported by the webserver is restricted only to static content, in which a client sends a request just to read a specific file from the server. To keep things simple, requests for dynamic content (such as output of an executable file) is not included in this assignment. But if you are interested in learning how dynamic content is handled, you may explore the original code of Prof. Remzi’s concurrent webserver project.

Each file on the server has a unique name known as URL (Universal Resource Locator) that is specified by user in a web client. For example, ‘http://localhost:8003/test1.html’. This URL denotes that HTTP protocol is to be used to fetch the file ‘/test1.html’ from the server that resides on the host machine ‘localhost’ and is accepting connections on port ‘8003’. Here, ‘localhost’ can be replaced by an ip address (like 173.194.217.102) or a domain name (like google.com) depending on the server we want to connect to, and the port can be omitted if the server is accepting connections on the default HTTP port (80). Please note that you may use this URL from any web browser on your machine, but not with the provided web client. The input format of the web client is slightly different (refer to section 3.1 for details).

The exact format of the content request from web client and the response from web server are out of the scope of this assignment. However, do note that the request handled by our web server uses the method ‘GET’ (as you can see from the provided code) and any other methods (e.g., POST) are not supported. Again, any interested student may refer to Prof. Remzi’s concurrent webserver project for other details regarding the HTTP request and response.

1. **Assignment problem statements**

In this assignment, you will need to modify the basic web server to add three pieces of functionality as explained below.

* 1. **Multi-threaded server**

The basic web server that we provided is a single threaded server (i.e. with main thread only, it does not spawn any new threads). So it suffers from the basic performance bottleneck issue as it can process only a single HTTP request at a time. Hence making this server multi-threaded is almost mandatory to make it a more complete web server.

The simplest approach to building a multi-threaded server is to spawn a new thread for every new HTTP request and the OS can schedule these threads according to its policy. However, the drawback of the one-thread-per-request approach is that the web server pays the overhead of creating a new thread on every request. Therefore, the generally preferred approach for a multi-threaded server is to create a fixed-size pool of worker threads when the web server is started. This pool of worker threads is created by the main thread and the size of the thread-pool will be accepted from the command line while starting the server. Each worker thread is blocked until there is an HTTP request for it to handle. Similar to single-threaded server, the HTTP requests are still accepted by the main thread, but it puts those requests in a buffer (preferably a queue) and returns to accepting more requests. The waiting worker threads will pick those requests from the buffer. Note that the master thread will not read the actual requested files, but it can do some initial processing like getting size of the requested file (that may be required for scheduling).

Here, you may have noticed that the main thread and worker threads are in producer-consumer relationship; hence their access to the shared buffer should be carefully synchronized. Also, the worker threads must be blocked if the buffer is empty. Also, there should be a maximum bound to the number of elements in the queue (which will also be accepted from the command line while starting the server). The main thread (which is accepting the HTTP requests) should be blocked when the buffer is full. To achieve the correct synchronization, you will need to use mutex and condition variables only. Do not use any other method. Also, there will be penalty for implementations with spin-waiting/busy-waiting.

* 1. **Scheduling policies**

The scheduling policy implemented by you will determine which HTTP request will be handled by the next thread (and **NOT** which thread will be scheduled at a given time – as that will be handled by the OS). You will need to implement two scheduling policies for the HTTP requests, as explained below.

1. **First-in-first-out (FIFO):** In this policy, the HTTP requests will be handled on the first come first serve basis. Note that the HTTP requests will not necessarily finish in FIFO order, as it depends upon how the OS schedules the active threads.
2. **Smallest-file-first (SFF):** In this policy, when a worker thread wakes, it handles the request for the smallest file. This policy approximates Shortest Job First to the extent that the size of the file is a good prediction of how long it takes to service that request. You will also note that the SFF policy requires that something be known about each request (e.g., the size of the file) before the requests can be scheduled. Thus, to support this scheduling policy, you will need to do some initial processing of the request (hint: using stat() on the filename) outside of the worker threads; you will probably want the master thread to perform this work, which requires that it read from the network descriptor. To implement this, you’ll need to carefully manage add and delete operations on your buffer.
   1. **Security**

Your system should make sure to constrain file requests to stay within the sub-tree of the file system hierarchy, rooted at the base working directory that the server starts in. You must take steps to ensure that pathnames that are passed in do not refer to files outside of this sub-tree. One simple (perhaps overly conservative) way to do this is to reject any pathname with ‘..’ in it, thus avoiding any traversals up the file system tree. More sophisticated solutions could use chroot() or Linux containers, but those are beyond the scope of this assignment.

1. **Implementation details**

Below subsections describe the code file details and the code modification details that you’ll have to follow.

* 1. Details of the code files

**wserver.c**: Contains main() for the web server and the basic serving loop.

**request.c**: Performs most of the work for handling requests in the basic web server. Start at request\_handle() and work through the logic from there.

**io\_helper.h and io\_helper.c**: Contains wrapper functions for the system calls invoked by the basic web server and web client. The convention is to add \_or\_die to an existing call to provide a version that either succeeds or exits. For example, the open() system call is used to open a file, but can fail for a number of reasons. The wrapper, open\_or\_die(), either successfully opens a file or exists (while printing the error message) upon failure.

**wclient.c**: Contains main() and the support routines for the very simple web client. To test if your server correctly handles concurrent requests as per the defined scheduling policy, you can launch wclient multiple times using a bash file and check the output (make sure your multiple requests are parallel, not sequential).

**Makefile**: We also provide you with a Makefile that can be used to create the executable files from the code (wserver and wclient). You can type “make” to create all of these programs. You can type “make clean” to remove the object files and the executables. You can type “make server” to create just the server program, etc.

After the compilation, you can use the format of the example commands given below to start the server and test with client.

./wserver –d ~/myServer –p 8003

./wclient localhost 8003 /test1.html

* 1. **Code modification details**

We recommend you to go through all the code files to understand how the server works. But if in case you are finding the flow too much complicated, you may read the comments in the files ‘wserver.c and request.c’ from the template and it should be easy enough to understand your tasks. Make sure you thoroughly understand at least these two files before starting to write the code. If some of you are still not able to understand, we’ll have a session to answer you queries and clear your doubts. **Again, note that** **you will have to modify only the ‘request.c’ file and submit that in gradescope**.

The main server file (wserver.c) from the template accepts various arguments from the command line (while starting the server). Your code must correctly use these arguments. Refer below for the details of those arguments.

./wserver [-d basedir] [-p port] [-t threads] [-b buffers] [-s schedalg]

**basedir:** this is the root directory from which the web server should operate. The server should try to ensure that file accesses do not access files above this directory in the file-system hierarchy. Default: current working directory (.).

**port:** the port number that the web server should listen on; the basic web server already handles this argument. Default: 10000.

**threads:** the number of worker threads that should be created within the web server. Must be a positive integer. Default: 4.

**buffers:** the number of request connections that can be added to the buffer at one time (must be a positive integer). Default: 64.

**schedalg:** the scheduling algorithm to be performed. Uses integer values 0 (for FIFO) and 1 (for SFF). Default: 0.

Finally, step-by-step add functionalities to your code. For example, you may want to start implementing a simple FIFO based multi-threaded server with no upper limit to buffer size, then add the upper limit to buffer size, then add support for SFF, then security part. Also make sure to keep a backup of your working code versions. If you have any doubts, you can reach out to me on teams.

**Important note**: *Add proper comments for every change that you make in the code and follow proper code indentation. As this assignment will be evaluated without your physical presence, it is your responsibility to make your code readable and avoid unnecessarily complex flows as lack of readability and clarity in understanding will ultimately cost marks. For this assignment too, we will try to autograde most part of the assignment along with code similarity check. So, do remember to follow strict output format. More details about that and submission instructions will be provided soon.*