# OS Lab CS232

## Winter 2021 LAB5

## Thread Pool Web Server

The goals of this project are:

* to understand the basic elements of the World Wide Web and its protocols.
* to learn how real-world multi-threaded servers are built.
* to apply scheduling algorithms to a working system.
* develop your skills in programming C and pthreads.
* gain experience in reading and modifying existing code.

### Introduction

Network servers of various kinds are the most likely place where you will encounter threads in real life. Because a network server must handle activities of multiple simultaneous users, it has many of the same structures and difficulties as an operating system. A good example of this is a web server, which must provide data access to many web browsers and other programs at once.

Let's briefly review how the web works. The web consists of servers and clients. A web server is a process that runs on a machine and makes data available to any client that may call it up and ask for it. A client that you are certainly familiar with is a web browser, but there are many other kinds of clients that interact with servers in quieter ways. For example, the wget command line tool can be used to request files from a web server without any of the graphical hoopla.

A file on a server is identified by a Uniform Resource Locator (URL) like http://www.iiitg.ac.in:80/index.html, which means to use the http protocol to talk to the machine www.iiitg.ac.in on port 80 to get the file named /index.html.

Most web applications use HTTP, the HyperText Transfer Protocol. This began life as a [simple protocol](http://www.w3.org/Protocols/HTTP/AsImplemented.html), but has become [very complex](http://www.w3.org/Protocols/rfc2616/rfc2616.html) over the years. The basic idea is this. First, the client connects to the server on the TCP port given by the URL. It then sends an HTTP request stating what file it wishes to retrieve, along with the version of the protocol that it understands.

GET /index.html HTTP/1.0

The server examines this request, and then sends a response header:

HTTP/1.1 200 OK

Date: Tue, 11 Jan 2021 21:31:45 IST

Server: Apache/1.3.27

Connection: close

Content-Type text/html

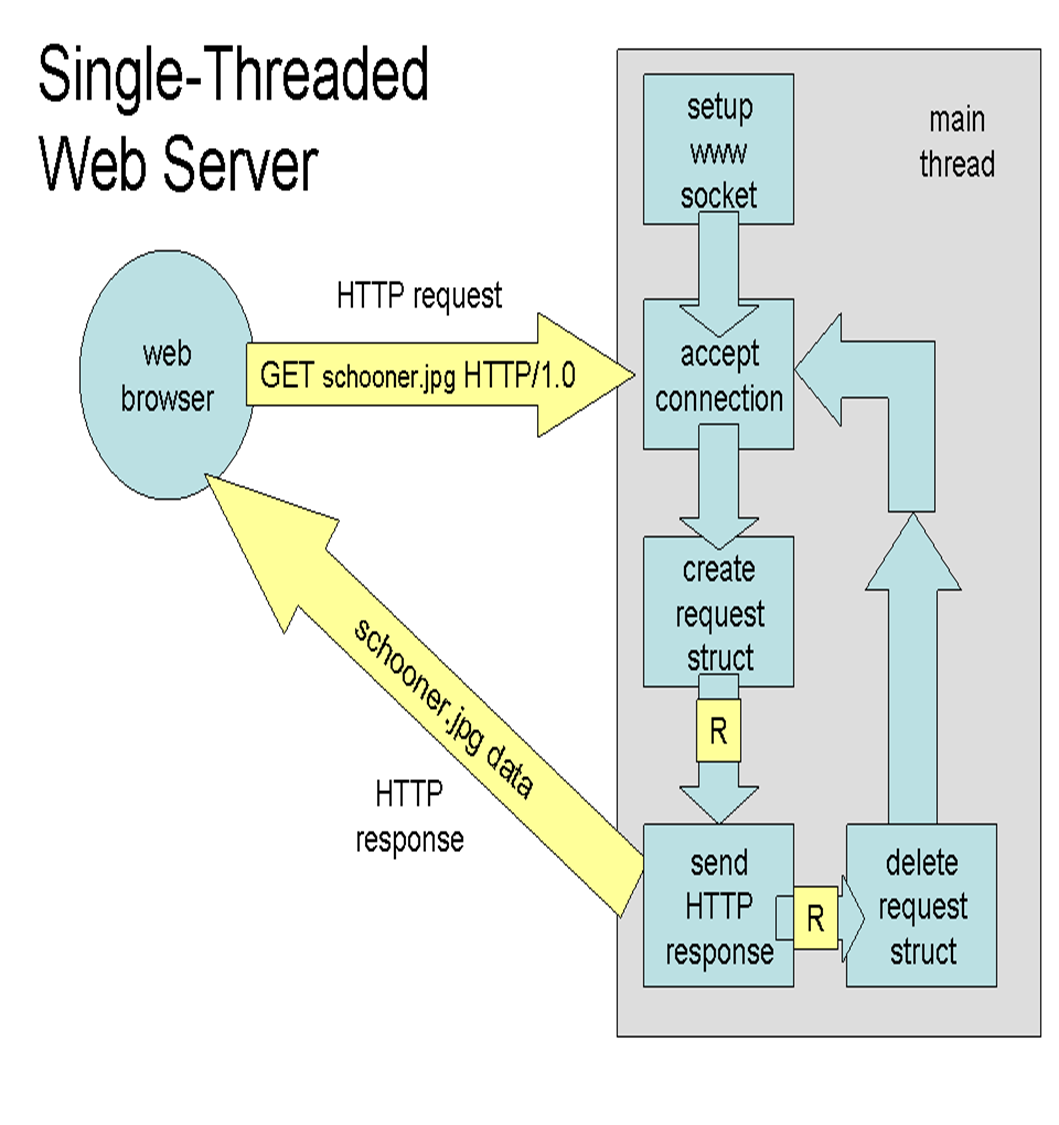
... followed by the actual data of the file in question. If you are curious, you can speak to web servers directly without an intervening browser by using the telnet tool. Try this to see the raw output of a web server:

% telnet www.cse.nd.edu 80

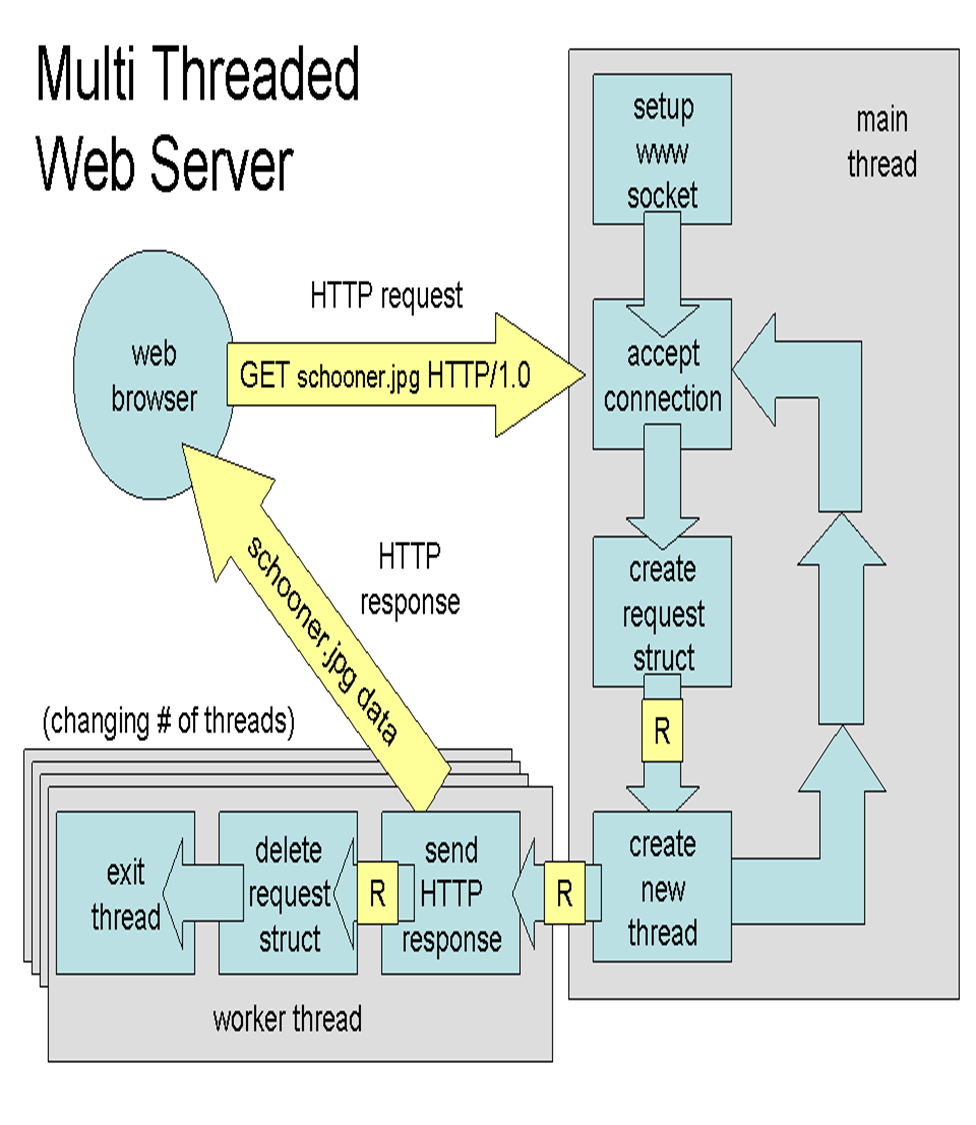
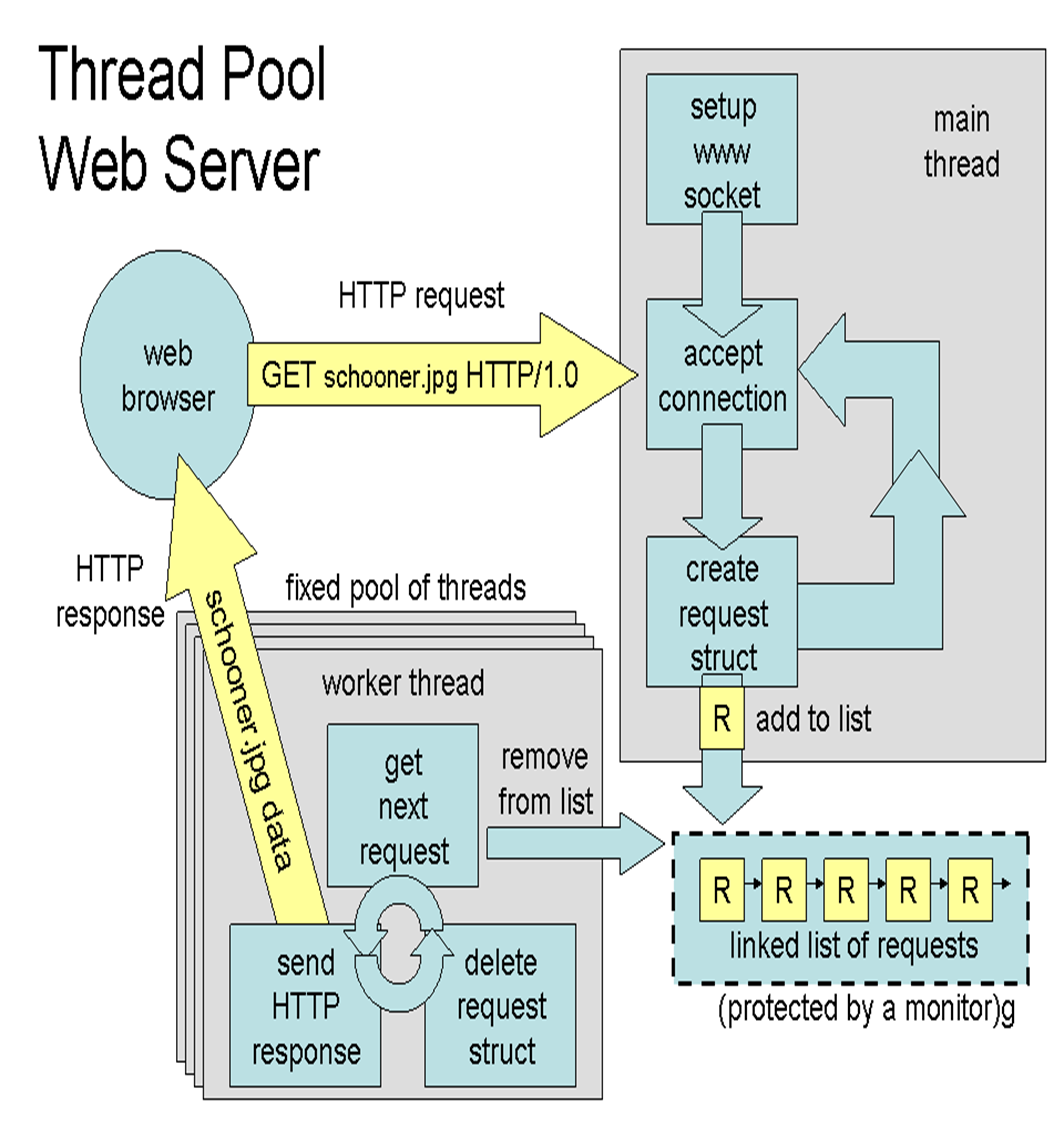
GET /index.html HTTP/1.0

(type return one more time)

Now that you know the basic underpinnings of the web, let's consider how one might build a web server designed to handle many incoming users. A good place to start is a single-threaded web server, shown to the right. In a single threaded web server, there is just one main loop. It sets up a listening socket, and then waits for incoming client connections. Once connected, it reads the HTTP request from the client and creates a "request" data structure that describes the caller and the nature of the request. Another routine services this request and sends back the HTTP response to the client. Once done, the main loop deletes the request and returns to waiting for incoming clients.

  
A single threaded web server is not likely to scale up to many simultaneous clients very well. The routine to send the HTTP response could be delayed for any number of reasons: the file system could be slow, the client could be unprepared to receive data, the network could be interrupted, or perhaps the response is dynamic and will take some time to produce. If this response is delayed, no other client will be able to obtain service until the blockage is repaired.

One response to this problem might be to create a new thread for every incoming connection. Such a multi-threaded web server is shown to the right. In a multi-threaded web server, the main loop listens for incoming client connections. When it discovers one, it reads the HTTP request and creates a request structure. It then creates a new thread and passes it the request structure. The new thread sends the HTTP response, deletes the request structure, and exits.

  
The multi-threaded web server will certainly scale better than a singly-threaded web server. If any server thread is delayed for any of the reasons mentioned above, new threads will still be created and the users will be happy. However, this design still has some problems. For starters, most thread packages are limited to a fixed maximum number of threads. If more clients than maximum threads arrive, the server is in trouble. Second, thread creation and deletion can be relatively expensive operations that are unnecessarily repeated under high load. Finally, a given machine may achieve optimum performance for a certain number of threads, independent of the number of actual clients. If we could control the number of threads without regard to the number of clients, the server can be tuned to maximum performance.   
  
So, most real-world servers use a thread-pool approach, shown to the right. In the thread-pool approach, the main thread creates a fixed number of worker threads. The main thread is still responsible for accepting connections. As it does so, it creates request objects and places them into a data structure such as a linked list or an array. Each worker threads pulls requests out of the buffer according to some scheduling algorithm, and then produces the necessary response. Thus, the main thread can accept connections as long as memory is available, while a fixed number of threads churn away at maximum efficiency.   


For this project, you will take a single-threaded web server and convert it into a thread-pool web server. As you probably have noticed, this is a real-life example of the producer-consumer problem, so you will need to use a monitor in order to protect the bounded buffer, making sure that each thread only proceeds when it is safe to do so. And, you will continue to gain practice in reading, modifying, and writing systems-level C code.   
  
Note that you **do not** write or modify any code that deals with sockets or the HTTP protocol: this isn't a networking class. Your job is to deal with the threading aspects, leaving the WWW aspects to the existing code.

### Getting Started

Begin with the source code given for a simple web server and the following use the webdocs package of files for testing. The provided code is a single-threaded web server. Build it, and then simply run webserver, choosing any port number above 1024 (USE 7XXX, where XXX are the last three digits of your roll number):

./webserver 7090

This simple web server will serve documents out of the webdocs directory, so make sure to unpack the webdocs package.

Now create a directory ~/lab5/test.

Open a new putty session, change directory to test and execute

wget localhost:7090/helmet-thumb.jpg

The file helmet-thumb.jpg should now be copied to directory test.

Now, create a directory test1 under test and from test execute

wget -P test1 -nd -p localhost:7090/index.html

All the files in webdoc will now get downloaded (why? Find out from man wget)

The provided web server is deliberately imperfect in a few different ways. First, each request to the server has a 33% chance of being delayed by 5 seconds. This will make it much easier to see the effect of using threads on human time scales. Second, the web server will automatically shut down if it does not receive any requests after five minutes. This automatic cleanup is in place to make sure that we don't end up with a lab full of idle web servers! Finally, the web server deliberately disables caching, so that your browser is forced to load everything when you hit reload, making it easy to test the system.

To see the limitations of a single-threaded server first-hand, try the following. Open up two putty sessions and issue

wget -P test1 -nd -p localhost:7090/index.html

wget -P test2 -nd -p localhost:7090/index.html

in session 1 and 2 respectively, simultaneously. You will notice that only one can make progress at a time: whenever one gets stuck, the other must wait as well.

### Technical Requirements

Create a directory lab5 in your home directory. Upload the webserver.zip file to this directory and unzip it. Then open up webdoc.tar.gz using tar.

Your job is to convert the given single-threaded web server into a thread-pool web server as described above. The total number of threads and the scheduling algorithm must be given on the command line. You should support a bounded buffer of up to ten requests. The main thread must block when the bounded buffer is full, and the worker threads must block if the bounded buffer is empty. Because the bounded buffer will be accessed by many threads simultaneously, you must protect it with a monitor as discussed in class.

Your web server must accept arguments as follows:

./webserver PORT NTHREADS MODE

Where PORT is the HTTP port number, NTHREADS is the number of threads in the pool, and MODE is the scheduling mode, which may be one of the following:

* **FCFS** (First Come, First Served) Each request is taken out of the buffer in the same order that it was inserted.
* **SFF** (Shortest File First) The smallest file is always taken out of the bounded buffer first.

You must come up with a testing method that demonstrates that your scheduling algorithms works correctly. Think carefully about how the thread-pool server should work and use some combination of printf in the server and creative use of the web browser to verify that your server works correctly. Make sure to test with both few and many threads, and with multiple web browsers simultaneously.

To accomplish this project, you may add or modify any code you see fit in webserver.c. You may **not** modify any other source code files, with the exception that you may add items to struct request in request.h if necessary.

### Hints

Use the stat system call to get information about files, including the file size.

Add a number of printfs to main to watch when clients connect to the server, when requests get created and serviced, and so on. This can help to clarify in what order things happen.

Note that buffer\_monitor should employ a monitor in order to synchronize the various threads. In POSIX threads, a monitor is constructed out of two separate data structures, a mutex (mutual exclusion) and a cond (condition variable).

You can use a browser as a client (instead of using wget) if you install Xming in your desktop / laptop and open an X window on the Amazon server. In that case you will use http://localhost:7XXX/filename to display a file in your borweser.

### Grading

**Note that busy waiting is not an acceptable solution to this assignment. If your solution uses busy waiting, you will receive zero credit for the synchronization component.**

Your grade will be based on:

* Correct synchronization, resulting in a safe program free of deadlocks, race conditions, and busy waiting. (40%)
* Correct implementation of each of the scheduling algorithms. (40%)
* Thorough attention to and handling of all possible error conditions, including user error. (10%)
* Good coding style, including clear formatting, sensible variable names, and useful comments. (10%)

### Turning In

### 10 / 11 March 2021 - implement a multithreaded server with new threads being created for each newly arriving request (see description above)

### 17 /18 March 2021 – implement a multi-threaded server with a pool of threads and the two scheduling policies as given above.

### If part 1 is not submitted by 10/11 March, only 80% marks will be given if the full system is submitted by 17 / 18 March.

Create webserver\_dynamic.c and webserver\_pool.c files for the two parts. Alter Makefile so that we get webserver\_dynamic and webserver\_pool as the executables

Submit a document detailing how you tested your implementation and the results you got. Specify how to carry out the tests so that we can evaluate. You can use documents other than those given in webdoc and you may want to create shell scripts to run your test.