P7: Web Server

Due Thursday by 11:59pm **Points** 100

In this assignment, you will be developing a real, working **web server.** To simplify this project, we are providing you with the code for a very basic web server. This basic web server operates with only a single thread. One of your main tasks is to make the web server multi-threaded so that it is more efficient. Your other main task is to create a shared memory segment which the threads can write to in order to export statistics that can be read by another process.

This project can be completed with a project partner.

Objectives

- To create and synchronize threads using pthreads, locks, and condition variables.
- To synchronize accesses to a shared buffer with threads in producer/consumer relationships.
- To understand the basics of a web server.
- To use shared memory across cooperating processes.
- To catch signals (such at SIGINT) with a signal handler.

Part A: Multi-Threaded Web Server

A.1 HTTP Background

Before describing what you will be implementing in this project, we provide a very brief overview of a simple web server and the HTTP protocol. Our goal in providing you with a basic web server is that you should be shielded from all of the details of network connections and the HTTP protocol. The code that we give you already handles everything that we describe in this section. If you are really interested in the full details of the HTTP protocol, you can read the **specification, (http://www.w3.org/Protocols/rfc2616/rfc2616.html)** but we do not recommend this for this project.

Most web browsers and web servers interact using a text-based protocol called HTTP (Hypertext Transfer Protocol). A web browser opens an Internet connection to a web server and requests some content with HTTP. The web server responds with the requested content and closes the connection. The browser reads the content and displays it on the screen.

Each piece of content on the server is associated with a file. If a client requests a specific disk file, then this is referred to as static content. If a client requests that a executable file be run and its output returned, then this is dynamic content. Each file has a unique name known as a URL (Universal Resource Locator). For example, the URL `www.cs.wisc.edu:80/index.html` identifies an HTML file called “index.html” on Internet host “www.cs.wisc.edu” that is managed by a web server listening on port 80. The port number is optional and defaults to the well-known HTTP port of 80. URLs for executable files can include program arguments after the file name. A “?” character separates the file name from the arguments and each argument is separated by a “&” character. This string of arguments will be passed to a CGI program as part of its “QUERY_STRING” environment variable.

An HTTP request (from the web browser to the server) consists of a request line, followed by zero or more request headers, and finally an empty text line. A request line has the form: `method uri version` . The `method` is usually GET (but may be other things, such as POST, OPTIONS, or PUT). The `URI` is the file name and any optional arguments (for dynamic content). Finally, the `version` indicates the version of the HTTP protocol that the web client is using (e.g., HTTP/1.0 or HTTP/1.1).

An HTTP response (from the server to the browser) is similar; it consists of a response line, zero or more response headers, an empty text line, and finally the interesting part, the response body. A response line has the form `version status message` . The `status` is a three-digit positive integer that indicates the state of the request; some common states are 200 for **OK** , 403 for **Forbidden** , and 404 for **Not found** . Two important lines in the header are **Content-Type**, which tells the client the MIME type of the content in the response body (e.g., html or gif) and **Content-Length** , which indicates its size in bytes.

Again, you don't need to know this information about HTTP unless you want to understand the details of the code we have given you. **You will not need to modify any of the procedures in the web server that deal with the HTTP protocol or network connections.**

A.2 Basic Web Server

The code for the web server is available from `~cs537-1/projects/web-server`. You should copy over all of the files there into your own working directory. You should compile the files by simply typing `make`. Compile and run this basic web server before making any changes to it! `make clean` removes .o files and lets you do a clean build.

When you run this basic web server, you need to specify the port number that it will listen on; you should specify port numbers that are greater than about 2000 to avoid active ports. When you then connect your web browser to this server, make sure that you specify this same port. For example, assume that you are running on royal-04.cs and use port number 2003; copy your favorite html file to the directory that you start the web server from. Then, to view this file from a web browser (running on the same or a different machine), use the URL: `royal-04.cs.wisc.edu:2003/favorite.html`. **Note that for some security reasons, your client (the browser) may need to be on the CS network to connect to your server. Given the current situation, you may not be able to physically present in the CS building, so we recommend using a CS department VPN.**

The web server that we are providing you is only about 200 lines of C code, plus some helper functions. To keep the code short and understandable, we are providing you with the absolute minimum for a web server. For example, the web server does not handle any HTTP requests other than GET, understands only a few content types, and supports only the QUERY_STRING environment variable for CGI programs. This web server is also not very robust; for example, if a web client closes its connection to the server, it may crash. We do not expect you to fix these problems!

The helper functions are simply wrappers for system calls that check the error codes of those system codes and immediately terminate if an error occurs. One should **always check error**

codes! However, many programmers don't like to do it because they believe that it makes their code less readable; the solution, as you know, is to use these wrapper functions. We expect that you will write wrapper functions for the new system routines that you call.

A.3 Overview: New Functionality

In this project, you will make the basic web server multi-threaded. You will also be modifying how the web server is invoked so that it can handle new input parameters (e.g., the number of threads to create).

The basic web server that we provided has a single thread of control. Single-threaded web servers suffer from a fundamental performance problem in that only a single HTTP request can be serviced at a time. Thus, every other client that is accessing this web server must wait until the current http request has finished; this is especially a problem if the current http request is a long-running CGI program or is resident only on disk (i.e., is not in memory). Thus, the most important extension that you will be adding is to make the basic web server multi-threaded.

The simplest approach to building a multi-threaded server is to spawn a new thread for every new http request. The OS will then schedule these threads according to its own policy. The advantage of creating these threads is that now short requests will not need to wait for a long request to complete; further, when one thread is blocked (i.e., waiting for disk I/O to finish) the other threads can continue to handle other requests. 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 first started. With the pool-of-threads approach, each thread is blocked until there is an http request for it to handle. Therefore, if there are more worker threads than active requests, then some of the threads will be blocked, waiting for new http requests to arrive; if there are more requests than worker threads, then those requests will need to be buffered until there is a ready thread.

In your implementation, you must have a **producer thread** (the main thread) that begins by creating a pool of worker threads, the number of which is specified on the command line. Your producer thread is then responsible for accepting new http connections over the network and placing the **descriptor** for this connection into a **fixed-size buffer;** in your basic implementation, the producer thread should not read from this connection. The number of elements in the buffer is also specified on the command line. Note that the existing web server has a single thread that accepts a connection and then immediately handles the connection; in your web server, this thread should place the connection descriptor into a fixed-size buffer and return to accepting more connections.

Each **worker thread** is able to handle both static and dynamic requests. A worker thread wakes when there is an http request in the queue. Once the worker thread wakes, it performs the read on the network descriptor, obtains the specified content (by either reading the static file or executing the CGI process), and then returns the content to the client by writing to the descriptor. The worker thread then waits for another http request.

Note that the producer thread and the worker threads are in a **producer-consumer** relationship and require that their accesses to the shared buffer be synchronized. Specifically, the producer thread must block and wait if the buffer is full; a worker thread must wait if the buffer is empty. In this project, you are required to use **condition variables**. **If your implementation performs any busy-waiting (or spin-waiting) instead, you will be heavily penalized.**

Side note: Do not be confused by the fact that the basic web server we provide forks a new process for each CGI process that it runs. Although, in a very limited sense, the web server does use multiple processes, it never handles more than a single request at a time; the parent process in the web server explicitly waits for the child CGI process to complete before continuing and accepting more http requests. **When making your server multi-threaded, you should not modify this section of the code.**

A.4 Web Server Specification

Your C program must be invoked exactly as follows:

```
prompt> server [port_num] [threads] [buffers] [shm_name]
```

The command-line arguments to your web server are to be interpreted as follows.

- **port_num:** the port number that the web server should listen on; the basic web server already handles this argument.
- **threads:** the number of **worker** threads that should be created within the web server. Must be a positive integer.
- **buffers:** the number of request connections that can be accepted at one time. Must be a positive integer. Note that it is not an error for more or fewer threads to be created than buffers.
- **shm_name:** the name of shared memory object (described more below)

For example, if you run your program as:

```
server 5003 8 16 shm-9452
```

then your web server will listen to port 5003, create 8 worker threads for handling http requests and allocate 16 buffers for connections that are currently in progress (or waiting); it will use `shm-9452` as the object name for the shared memory segment. Because everyone will create the file at `/dev/shm/`, you have to use a unique file name to avoid conflicts with other people using the same CSL machine. Therefore, when testing your code, we recommend using a string that is less likely to conflict e.g. your cs_login or some random number.

Once your server is running, you can make a request to it my runinng the client program. If you are on the same lab machine you can simply use localhost as the host:

`client localhost 5003 /home.html`

`client localhost 5003 /output.cgi?2`

Note in this case, you don't need to use the CS department VPN because CSL machines are already on the CS network.

A.5 Web Server Hints

We recommend understanding how the code that we gave you works. We provide the following files:

- **server.c:** Contains main() for the basic web server.
- **request.c:** Performs most of the work for handling requests in the basic web server. All procedures in this file begin with the string "request".
- **helper.c:** Contains wrapper functions for the system calls invoked by the basic web server and client. The convention is to capitalize the first letter of each routine. Feel free to add to this file as you use new libraries or system calls. You will also find a corresponding header (.h) file that should be included by all of your C files that use the routines defined here.
- **client.c:** Contains main() and the support routines for the very simple web client. To test your server, you may want to change this code so that it can send simultaneous requests to your server. At a minimum, you will want to run multiple copies of this client.
- **output.c:** Code for a CGI program. Basically, it spins for a fixed amount of time, which you may useful in testing various aspects of your server.

We also provide you with a sample Makefile that creates server, client, and output.cgi. 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. As you create new files, you will need to add them to the Makefile.

The best way to learn about the code is to compile it and run it. Run the server we gave you with your preferred web browser. Run this server with the client code we gave you. You can even have the client code we gave you contact any other server that speaks HTTP. Make small changes to the server code (e.g., have it print out more debugging information) to see if you understand how it works. Note that your client (the browser) may need to be on the CS network to connect to your server.

We anticipate that you will find the following routines useful for creating and synchronizing threads: `pthread_create`, `pthread_mutex_init`, `pthread_mutex_lock`, `pthread_mutex_unlock`, `pthread_cond_init`, `pthread_cond_wait`, `pthread_cond_signal.` To find information on these library routines: read the man pages. You should also feel free to read the OS book, which contains a great amount of detail on how to build producer-consumer relationships between threads.

Part B: Shared Memory for Statistics

In this part of the project, you will enable your multi-threaded web server to communicate with another process (we call it "stat process") through a **shared memory segment** to display statistics about the web-server threads. These two processes will run on the same machine. Each web-server thread will periodically **write** to the shared memory segment with updates about its recent behavior; the **stat process** collects this information (by **reading** from the shared memory segment) and periodically displays the information.

You will need to ensure that your web-server **threads** do not modify each other's data in the shared segment, but you do not need to synchronize across the two **processes**; if the stat process happens to occasionally read data that is not up-to-date or not fully consistent, that is okay, since the data is just usage statistics (and not your bank account).

B.1 Web server producer thread: create and delete shared memory segments

It is the responsibility of the producer thread of the web server process to create and initialize the shared memory segment.

1. Create a shared memory object. A shared memory segment can be created with `shm_open()`. Each shared memory segment is identified with a unique shared memory object name; this object name is used to match requests for a particular segment between the server and the clients. You should obtain this object name from the command line. To create a shared memory object with the name `SHM_NAME`:

```
int shm_fd = shm_open(SHM_NAME, O_RDWR | O_CREAT, 0660);
```

If successful, this call will create a "fake" file `/dev/shm/SHM_NAME` and return the file descriptor for this file. The rest of the operation will perform on this file/fd. `O_RDWR | O_CREAT` means this shared memory should be both readable and writable and will be created if not exists. 0660 is the filesystem permission if this file is newly created. You will need to read the man page of `shm_open()` to understand the other details.

2. Initialize the shared memory object. A newly created shared memory object will have zero sizes (contains no data). You need to call `ftruncate()` to resize this file to a precise size in bytes (if the filesize grows, `ftruncate()` will initialize new space with zero). For this project, you must only create **one page** of shared memory. Don't guess or hard-code this value; figure out how you can find the right page size at run-time instead. (Hint: find the page size using some system call). Again, you should read the man page of `ftruncate()` to understand the details.

3. Map the shared memory object into the address space. You should use `mmap()` and the argument flags in this routine should be `MAP_SHARED` because we are sharing this mapping with other processes. For example,

```
void* shm_ptr = mmap(NULL, PAGESIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
```

This will map the shared memory object into the address space and return the pointer to the mapped area. In other words, `[shm_ptr, shm_ptr + PAGESIZE)` is the shared memory segment. To make web server worker thread easier to use this shared memory segment, you may need to cast this page of memory into an array and each worker takes one slot. For example.

```
typedef struct {
    // ...
} slot_t;
slot_t* shm_slot_ptr = (slot_t*) shm_ptr;
```

You can then use the index to access a specific slot e.g. to access the fifth slot: `shm_slot_ptr[5]`. You must be able to handle up to 32 worker threads, so make sure the structure array you define will fit within a single page.

If anything goes wrong with this setup (e.g., the shared memory page cannot be exclusively created), then the `shm_server` should exit with return code 1.

4. Delete the shared memory object. How will you know when your web server is terminating? We assume that you will kill your web server program by sending it a SIGINT signal (with Ctrl-C). Usually, SIGINT interrupts your program and kills it. However, you can change this default behavior by specifying a signal handler that should be run when that particular signal is delivered. To do this, use `signal()` to specify the routine that you want to run. This new routine should unmap and delete the shared memory segment and then exit with status 0. To do so, you need to call `munmap()` and `shm_unlink()`. Again, read the manuals carefully to learn how to correctly use them!

B.2 Web server worker thread: write to the shared memory

Whenever one of your worker threads finishes an HTTP request, it should update its statistics stored in shared memory. Specifically, for each thread, you should track its thread id (obtained from `pthread_self()`), the total number of HTTP requests it has completed thus far, the number of static requests, and the number of dynamic requests).

To make sure worker threads won't overwrite each other's data, you should assign each worker thread a slot and it only writes to its own slot. Again, we don't require the synchronization across the web server process and stat process, so it is okay if the stat process is reading while a web server thread is writing.

B.3 Stat process: read from the shared memory

You will need to create a separate stat process to read and display your statistics. The stat process doesn't need to create/truncate/initialize a shared memory object, but simply map the existing shared memory object and read from it. After mapping the shared memory, your stat process will iterate between sleeping and printing **forever** until killed by SIGINT. Note that you don't need to handle the signal in the stat process because it performs read-only job. It's okay to just leave it killed.

This process, `stat_process` should take the following arguments:

```
stat_process [shm_name] [sleeptime_ms] [num_threads]
```

- `shm_name`: the shared memory object name (the same one passed to the web server)
- `sleeptime_ms`: Each iteration, the process should sleep for sleeptime_ms (milliseconds). The function `nanosleep` is good for this functionality. When it wakes, it should read the shared statistics for every thread and print to stdout.
- `num_threads`: the number of worker threads on the server (so the number of valid segments in the shared memory).

If run incorrectly, it should print an error message and exit with return code 1. If the shared memory segment does not exist, it should print an error message and exit with return code 1.

For example:

```
stat_process shm-9452 1000 2
```

will read and print stats every 1000 ms (i.e., every 1 second).

Every iteration (after it has slept 1 time), it should print the following information exactly:

```
\n<Iteration i>
<TID t1> : <Requests t1> <Static t1> <Dynamic t1>
<TID t2> : <Requests t2> <Static t2> <Dynamic t2>
```

For example, the following is the possible output:

```
1
140301066589952 : 20 15 5
140301058197248 : 11 9 2

2
140301066589952 : 22 16 6
140301058197248 : 15 9 6
```

Tips

1. Make sure you include proper headers to use `shm_open()` and `mmap()`. **To use `shm_open()`, you need to use `-lrt` during compilation. `rt` is a library for real-time-related routines**; prefix with `-l` is to tell the compiler that you want to link with this library.

2. In addition to the provided client program, you can use other clients such as netcat or curl to test the server. Netcat allows you to send data over the connection at the exact time you want to. This helps when testing your buffer implementation. Here is an example netcat command: `nc localhost 8080 -C` While this command is running, you can type a request (e.g. `GET / HTTP/1.0`) and then type enter twice and you should receive a response from the server.

Code Delivery

Handing in Your Code

EACH project partner should turn in their joint code to each of their handin directories.

So that we can tell who worked together on this project, each person should place a file named **partners.txt** in their handin/p7 directory. The format of **partners.txt** should be exactly as follows:

```
cslogin1 wisclogin1 Lastname1 Firstname1
cslogin2 wisclogin2 Lastname2 Firstname2
```

It does not matter who is 1 and who is 2. If you worked alone, your **partners.txt** file should have only one line. There should be no spaces within your first or last name; just use spaces to separate fields.

To repeat, both project partners should turn in their code and both should have this **partners.txt** file in their handin/p7 directory.

Within your p7 directory, make the following directories and place your code in them as follows:

```
~cs537-1/handin/<login>/p7/ontime/<web server files>
```

If you wish to use slip days in this project, then you should submit your code to the corresponding slip directory: **slip1**, **slip2**, or **slip3. slip1** indicates that you wish to use one slip day in this project. We will use the latest submission for grading. This means that if you submit code in folders slip3 and slip1, then we will grade the version submitted at slip3.

Will need to modify `Makefile` to compile, create, and clean `stat_process`

Testing

Ensure correct behavior from these tests before using the tester. The tester is at `~cs537-1/tests/p7/run-tests.sh` If you want to run just test n, you can run `~cs537-1/tests/p7/run-tests.sh -t n` If you want to view a list of the tests you can run `~cs537-1/tests/p7/list-tests.sh` Note that there will be a small number of hidden test cases (25%).

We added a debug mode so that you can see the server's `stdout` and `stderr` in the `tests-out` folder. Running debug mode will usually cause the tests to fail, so if you want to actually see if you pass the tests, then don't use debug mode. You can use debug mode by setting the `TEST_DEBUG_MODE` environment variable when running the tests: `TEST_DEBUG_MODE=1 ~cs537-1/tests/p7/run-tests.sh`

Slip Day Policy

A maximum of 3 slip days can be used for this project no matter you are working with a partner or not. Additional 2 slip days for each one have been added as described in this [post (https://piazza.com/class/kjn4sz4kq7t2d2?cid=596)](https://piazza.com/class/kjn4sz4kq7t2d2?cid=596) .

If you are working with a partner, then

- Each of you will only need to contribute 1/2 of any slip days you use; for example, if you use 1 slip day, each of you is charged 1/2 of a day.
- If only one of you runs out of slip days, the needed slip days will be taken from the partner who still has them.
- A 1/2 slip day can't be used (unless you are combining with a 1/2 slip day from your partner). We will assume you are aware of your partner's slip days and the implications.