Programming Project

Inter Process Communication and Synchronization :

Versão 1 31/3

Web Servers

The main objectives of this project are

- To modify an existing code base.
- To understand inter process communication.
- To learn how to create and synchronize cooperating threads in Linux.
- To understand how a basic web server is structured.
- To consider hoew to test the functionality of an application.

Notes

- When compiling and linking - use the coorect librray arguments for example for the posix pthread librray use **-lpthread**
- It is OK to talk to others in class about what you are doing. It is even OK to explain to someone how some particular things works (e.g., how `pthread_create()` works). However, it is **NOT OK** to share code. If you are having trouble, **come talk to us instead.**
- **Use git (gitlab ou gitcom or your own private git server)**

Background

In this project you will be developing a **web server**

In this project you will start with the code for a basic web server. (tiny)

The code is from the book : Computer Systems: A Programmer's Perspective, 3/E

Reading the book is ESSENTIAL

The original code can be obtained from the site : http://csapp.cs.cmu.edu/3e/students.html

Howver I suggest that you use **my initial version on Moodle** which has some compiler warnings removed and Maklefile. There is also vscode debug and launch json for windows wsl using the gdb toolchain as well as a client and crypto library with example.

This basic web server operates with only a single thread; it will be your main job to make the web server multi-threaded so that it is more efficient.

You will be required to priovide a Relatory with diagrams of the functioning of the web server (flow chart with forks etc). More details later

HTTP Background

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.

- static content
    - A client requests a specific disk file which should be returned with the approriate file type (text/json/video etc)
- dynamic content
    - A client requests that a executable file be run and its output returned, this is

Each file has a unique name : a URL (Universal Resource Locator).

For example, the URL www.ubi.pt:80/index.html identifies an HTML file called "/index.html" on Internet host "www.ubi.pr" 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 "query parameters" after the file name.

- A '?' character separates the file name from the arguments
- 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 the verb 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 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.

If you would like to see the HTTP protocol in action, you can connect to any web server using `telnet`. For example, run `telnet www.di.ubi.pt 80` and then type (note that there is an empty line at the end):

```
GET / HTTP/1.1
host: www.di.ubi.pt

```

You will then see the HTML text for that web page!

Basic Web Server

The code for the web server is available from `the book site or from Moodle.`

Your group 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.

I have also included a vscode build and debug config file on Moodle

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 1024 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 192.168.1.33 and use port number 8080; copy your favorite.html file to the directory that you start the web server from.

To view this file from a web browser (running on the same or a different machine), use the url:
```
sala619-1.di.ubi.pt:8080/favorite.html
```

To view this file using the client code I am giving you, use the command
```
client sala619-1.di.ubi.pt 8080  /favorite.html
```

To run the cgi program, adder.cgi, using the client code and sending the arguments 1, 5, 1000, and 0, use the command
```
client  sala619-1.di.ubi.pt 8080 "/adder.cgi?1&5&1000&0"
```

The web server that we are providing you is only about 200 lines of C code, plus some helper functions. 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 (e.g., if the user presses the "stop") 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 programmer don't like to do it because they believe that it makes their code less readable. The solution is to use these wrapper functions.

Note the common convention that we use of naming the wrapper function the same as the underlying system call, except capitalizing the first letter, and keeping the arguments exactly the same. **We expect that you will write any wrapper functions for the new system routines that you call.**

Overview: New Functionality

In this project, you will be adding functionality to both the web server code and the web client code.

You will be adding three key pieces of functionality to the basic web server.

- You will add a new cgi-bin function. You can usa as a model the existing adder function. This new function will be called proofofwork and accepts a string and difficulty number (n) and attempts to find a nonce (an integer) such that
  - HASH(string || atoi(nonce)) = 20 bytes where the first n bytes are zeros
  - It then returns the nonce and hash
  - Thus its simple (and rapid) for the client to check that the server did this work.
- You will make the web server multi-threaded, with the appropriate synchronization.
- You will implement different scheduling policies so that requests are serviced in different orders.
- You will add statistics to measure how the web server is performing.

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).

You will also be adding functionality to the web client for testing.

You should think about how this new functionality will help you test that the web server is implemented correctly. You will modify the web client so that it is also multi-threaded and can initiate requests to the server in different, well-controlled groups.

Aptitude / Warm Up

You shoud complete some of the exercices from the book – chapter 12 – for the Multi Processing version (fork) as you feel comfortable with

- Implement more file types (12.7) e.g mp4/video
- Write form for adder and check it with a real browser (12.10)
- Implement more cgi-bin functions in particular a proof of work function and write a new client to test it.
- Implement the server as multiprocessed (while (accept fork / wait)) with the signal handling. This is exactly the same as in the example programs included on Moodle.
- Keep this version as "Multi Processed Version" safe for inspection before removing this section of code to cokmplete the main task of a "Multithread Version"

Part 1: Multi-threaded Server

The basic web server 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 (such as the proof of work 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.

You should implement this simple change before procedding.

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 master thread that begins by creating a pool of worker threads, the number of which is specified on the command line. Your master 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 master 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. You should investigate how to create and manage posix threads with `pthread_create` and `pthread_detach`.

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; when there are multiple http requests available, which request is handled depends upon the scheduling policy, described below.
- 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 master thread and the worker threads are in a producer-consumer relationship and require that their accesses to the shared buffer be synchronized.

Specifically,

- the master 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 advised to use `condition variables`.

Avoid any busy-waiting (or spin-waiting) instead.

Side note: Do not be confused by the fact that the basic web server provided forks a new process for each CGI process that it runs. Although, in a very limited sense, the BASIC 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.**

Part 2: Scheduling Policies

In this project, you will implement a number of different scheduling policies. Note that when your web server has multiple worker threads running (the number of which is specified on the command line), you will not have any control over which thread is actually scheduled at any given time by the OS. Your role in scheduling is to determine which http request should be handled by each of the waiting worker threads in your web server.

The scheduling policy is determined by a command line argument when the web server is started and are as follows:

- `Any Concurrent Policy (ANY):` When a worker thread wakes, it can handle any request in the buffer. The only requirement is that all threads are handling requests concurrently. (In other words, you can make ANY=FIFO if you have FIFO working.)
- `First-in-First-out (FIFO):` When a worker thread wakes, it handles the first request (i.e., the oldest request) in the buffer. Note that the http requests will not necessarily finish in FIFO order since multiple threads are running concurrently; the order in which the requests complete will depend upon how the OS schedules the active threads.
- `Highest Priority to Static Content (HPSC):` When a worker thread wakes, it handles the first request that is static content; if there are no requests for static content, it handles the first request for dynamic content. Note that this algorithm can lead to the starvation of requests for dynamic content.
- `Highest Priority to Dynamic Content (HPDC):` When a worker thread wakes, it handles the first request that is dynamic content; if there are no requests for dynamic content, it handles the first request for static content. Note that this algorithm can lead to the starvation of requests for static content.

You will note that the HPSC and SPDC policies require that something be known about each request before the requests can be scheduled. Thus, to support this scheduling policy, you will need to do some initial processing of the request outside of the worker threads; you will want the master thread to perform this work, which requires that it read from the network descriptor.

Part 3: Usage Statistics

You will need to modify your web server to collect a variety of statistics. Some of the statistics will be gathered on a per-request basis and some on a per-thread basis. All statistics will be returned to the web client as part of each http response. Specifically, you will be embedding these statistics in the entity headers; we have already made place-holders in the basic web server code for some of these headers. You should add the additional statistics. Note that most web browsers will ignore these headers that it doesn't know about; to access these statistics, you will want to run our modified client.

For each request, you will record the following counts or times; all times should be recorded at the granularity of milliseconds. You may find **gettimeofday**() useful for gathering these statistics. This function gets the current time is seconds and microseconds !

- `Stat-req-arrival-count:` The number of requests that arrived before this request arrived. Note that this is a shared value across all of the threads.
- `Stat-req-arrival-time:` The arrival time of this request, as first seen by the master thread. **This time should be relative to the start time of the web server.**
- `Stat-req-dispatch-count:` The number of requests that were dispatched before this request was dispatched (i.e., when the request was picked by a worker thread). Note that this is a shared value across all of the threads.
- `Stat-req-dispatch-time:` The time this request was dispatched (i.e., when the request was picked by a worker thread). **This time should be relative to the start time of the web server.**
- `Stat-req-complete-count:`
  - For **static** content, the number of requests that completed before this request completed; we define completed as the point after the file has been read and just before the worker thread starts writing the response on the socket.
  - For **dynamic** requests (that is, CGI scripts), you do not need to update or return this count.
  - Note that this is a shared value across all of the threads.
- `Stat-req-complete-time:` For static content, the time at which the read of the file is complete and the worker thread begins writing the response on the socket. For dynamic requests (that is, CGI scripts), you do not need to return this time. (But it is possible !))
  - **This time should be relative to the start time of the web server.**
- `Stat-req-age:` The number of requests that were given priority over this request (that is, the number of requests that arrived after this request arrived, but were dispatched before this request was dispatched).

You should also keep the following statistics for each thread:

- `Stat-thread-id:` The id of the responding thread (numbered 0 to number of threads-1)
- `Stat-thread-count:` The total number of http requests this thread has handled
- `Stat-thread-static:` The total number of static requests this thread has handled
- `Stat-thread-dynamic:` The total number of dynamic requests this thread has handled

Thus, for a request handled by thread number i, your web server will return the statistics for that request and the statistics for thread number i.

- All time statistics (i.e., stat-req-arrival-time, stat-req-dispatch-time, and stat-req-complete-time) should be relative to the time (in ms) at which the web server was started. (Store the start time in a **global variable** so that it is simple to subtract it from all of your other times.)

Part 4: Multi-threaded Client Can be developed independently of the server/statistics

We provide you with a basic single-threaded client that sends a single HTTP request to the server and prints out the results. This basic client prints out the entity headers with the statistics that you added, so that you can verify the server is ordering requests as expected.

While this basic client can help you with some testing, it doesn't stress the server enough with multiple simultaneous requests to ensure that the server is correctly scheduling or synchronizing threads. Therefore, you need to modify the client to send more requests with multiple threads. Specifically, your new client must implement two different request workloads (specified by a command line argument you will add).

All versions take a new command line argument: N, for the number of created threads.

You can define a maximum value (e.g 20) to simplify your program e.g pthread_t threads[MAX] - if not then be prepared to use dynamic variables for threads, semaforos etc for exasmple pthread_t *threads=(malloc (N * sizeof ..) followed by initialization loops when necessary)

- `Concurrent Groups (CONCUR)`: The client creates N threads and uses those threads to concurrently (i.e., simultaneously) perform N requests for the same file; this behavior repeats forever (until the client is killed). You should ensure that the N threads overlap sending and waiting for their requests with each other. After all of the N threads receive their responses, the procedure should repeat or the threads should repeat the requests.

  Two Options for implementation

  (1) Create and Wait for the N threads in a while Loop – this is the easiest

```
while (1) {
    For Loop N times
         Create thread that each send and receive one request
     For Loop n times
         pthread_wait  thread
  }
```

  (2) Create and Wait for N threads but the forever loop is inside the threads

  You may find the routine `pthread_barrier_wait` useful for implementing this; in no case should busy-waiting be used.

```
For Loop N times
      Create thread for a request

Each thread does
    While (1) {
          Make and receive one request
          Wait for barrier
}
```

- `First-In-First-Out Groups (FIFO)`: The client creates N threads and uses those threads to perform N requests for the same file; however, the client ensures that the requests are initiated in a serial order, but that the responses can occur in any order. Specifically, after one thread sends its request, it should signal to another thread that it can now send its request, and so on for the N threads; the N threads then concurrently wait for the responses. After all of the N threads receive their responses, the threads should repeat the requests until the client is killed.
    - You might find semaphores useful for implementing this behavior; in no case should busy-waiting be used.

```
Exemplo 5 threads são criados

   Criar 5 semaforos

   • sem_init semaforo numero  0 -> 1 (open)
   • sem_init  semaforo numero 1,2,3,4 -> 0 (closed)

   Para Cada thread ID

   • Sem wait Semaforo ID
   • Critical section (fazer pedidio)
   • Sem post  Semaforo ID+1 ( dizer ao proximo para avançar)
```

I suggest that you REFORMULATE the client code in functional sections - that way it's easy to implement multi-threading etc. Ideas below :

Use GLOBAL variables that will have the same values across all the threads : Exampes : hostent, protoent, hostname, port etc.

Create a Function httpProtocol that does the : Connect, Send Request , Read Reply etc

Program Specifications

For this project, you will be implementing both the server and the client.

Your web server must be invoked exactly as follows:

```
server [portnum] [threads] [buffers] [schedalg]
```

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

- `portnum:` 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 less threads to be created than buffers.
    - Exemple: we have 5 worker threads and accept a maximum of 25 connections
    - Exemple: we have 5 worker threads and accept a maximum of 2 connections
- `schedalg:` the scheduling algorithm to be performed. One of ANY, FIFO, HPSC, or HPDC.

For example, if you run your program as `server 5003 8 16 FIFO`

then your web server will listen to port 5003, create 8 worker threads for handling http requests, allocate 16 buffers for connections that are currently in progress (or waiting), and use FIFO scheduling for arriving requests.

Your web client must be invoked exactly as follows:

```
client [host] [portnum] [threads] [schedalg] [filename1] [filename2]
```

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

- `host:` the name of the host that the web server is running on; the basic web client already handles this argument.
- `portnum:` the port number that the web server is listening on and that the client should send to; the basic web client already handles this argument.
- `threads:` the number of threads that should be created within the web client. Must be a positive integer.
- `schedalg:` the scheduling algorithm to be performed. Must be one of CONCUR or FIFO.
- `filename1:` the name of the file that the client is requesting from the server.
- `filename2:` the name of a second file that the client is requesting from the server. This argument is optional. If it does not exist, then the client should repeatedly ask for only the first file. If it does exist, then each thread of the client should alternate which file it is requesting.

TimeofDay Example

Compile with -lm

```
int main() {
  struct timeval start, stop;
  gettimeofday(&start, NULL);
  printf("seconds : %ld micro seconds : %ld \n", start.tv_sec, start.tv_usec);

  //WORK
 //sleep(1);
 for (float i=0;i<600000;i=i+0.033) sum+=sqrt(i)+sqrt(i/2);

  gettimeofday(&stop, NULL);
  printf("seconds : %ld micro seconds : %ld \n", stop.tv_sec, stop.tv_usec);

  long secs = stop.tv_sec -start.tv_sec;
  long ms   = stop.tv_usec - start.tv_usec;
  if (ms<0){
		secs--;
		ms=1e+6+ms;
  }

   printf("Elapsed: seconds : %ld micro seconds : %ld\n", secs, ms);

  //total microseconds is secs*1e+6 + ms

  printf("%f\n",sum);
  return 0;
}
```