CSci 4061: Introduction to Operating Systems

Assignment 4: Multi-Threaded Web Server with Caching

Due: Nov 30th 2pm, you may work in a group of 2 or 3.

1 Overview

The purpose of this lab is to construct a multithreaded web server using POSIX threads (pthreads) in C to learn about thread programming and synchronization methods. Your web server will be able to handle any type of files HTML, GIF, JPEG, TXT, etc. It will handle a limited portion of the HTTP web protocol (namely, the GET command to fetch a web page). We will provide pieces of code (some already compiled into object files, and some source) that will help you complete the lab. ***Do not be overwhelmed - we have simplified this assignment for you. Read the whole specification before starting.***

2 Description

Your server will be composed of three different types of threads: dispatcher threads, worker threads and prefetcher threads. The purpose of the dispatcher threads is to repeatedly accept an incoming connection, read the request from the connection, and place the request in a queue for a worker thread to pick up and serve, in a loop. We will assume that there will only be one request per incoming connection. The purpose of the worker threads is to monitor the request queue, pick up requests from it and to serve those requests back to the client. Worker threads also put those requests back to another prefetching queue which will be read by the prefetcher threads. Prefetcher threads will then check if they can guess what the next request is going to be and bring the guessed page from the disk to the cache. Be careful to synchronize these two bounded queues.

You will use the following functions which have been pre-compiled into object files that we have provided:

```
void init(int port);    // run ONCE in your main thread

// Each of the next two functions will be used in order, in a loop, in the dispatch threads:
int accept_connection(void);
int get_request(int fd, char *filename);

// These functions will be used by the worker threads after handling the request
int return_result(int fd, char *content_type, char *buf, int numbytes);
(or int return_error(int fd, char *buf);)
```

```
// For the prefetcher thread
int nextguess(char *filename, char *guessed);
```

Full documentation of these functions has been provided in util.h. More will be said below about how to use these functions.

3 Incoming Requests

An HTTP request has the form: GET /dir1/subdir1/.../target file HTTP/1.1 where /dir1/subdir1/.../ is assumed to be located under your web tree root location. Our get request function automatically parses this for you and gives you the /dir1/subdir1/.../target file portion and stores it in filename parameter. Your web tree will be rooted at a specific location, specified by one of the arguments to your server. For example, if your web tree is rooted at /home/user/joe/html, then a request for /index.html would map to /home/user/joe/html/index.html.

4 Caching and Returning Results

Your web server will implement caching, a technique that greatly improves performance for frequently fetched pages and files. This cache stores (request, result) entries, where result is the actual data (in memory) of the file to be returned. The web cache size (number of entries) is defined by an argument (you may assume a maximum of 100 cache entries) and new entries may over-write old ones. If a request is not in the cache, the worker will fetch the file contents from disk and then put the new mapping (request, result) in the cache. When the file contents are fetched, you will also need to determine the file size.

Prefetching: Prefetching is another way to improve client experience. Prefetcher will try to make a guess for the next file that the client might request for and bring that file to the cache if it is not already there. The benefit of this is that if the client actually makes that request the server can very quickly serve it from its cache. You will assume that the prefetching queue can store 100 requests. If the prefetching queue is full, worker threads will neither try to add new requests to it nor wait for the queue to be non-full. Worker threads will add new requests only when the prefetching queue is not full. This is because compared to other tasks the server performs, prefetching is a low priority task.

Prefetcher threads repeatedly check this queue, remove the next request, make a guess and if that guessed file is not already in the cache, bring it to the cache. You will use **nextguess** to guess the next file based upon current request. This function will return 0 and set the guessed filename if it can find a file that needs to prefetched otherwise it will return a nonzero integer. If a guess is made you will need to store the guessed file to your cache as you do with the actual request.

For now, you will use one replacement strategy when the cache is full: Replace the oldest entry in the cache with the new cache entry. Again, since multiple threads may be accessing the cache, you must synchronize access to it. You will use **return_result** to send the data back to the web browser from the worker threads provided the file was opened and read correctly, or the data was found in the cache. If there was any problem with accessing the file, then you should use **return_error** instead. Our code will automatically append HTTP headers around the data before sending it back to the browser. Part of returning the result is sending back a special parameter to the browser: the content-type of the data. You may make assumptions based on the extensions of the files as to which content-type they are:

- Files ending in .html or .htm are content-type "text/html"
- Files ending in .jpg are content-type "image/jpeg"
- Files ending in .gif are content-type "image/gif"
- Files that have not been classified in the above categories may be considered, by default, to be of content-type "text/plain".

5 Modes of Operation

You need to implement three modes of operation for the web server.

First Come First Served (FCFS) – Each request is taken out of the request buffer by the worker in the same order it was inserted by the dispatcher.

Cached Requests First (CRF) – The worker searches the cache to see whether any of the requests can be satisfied from the cache. If there are no cached requests, the server falls back to FCFS.

Smallest File First (SFF) – Smallest file is always taken out of the request queue first, by the worker. Note that this is irrespective of whether the file is in cache or not. You may use **stat** system call to get the size of the file.

6 Request Logging

From the worker threads, you must carefully log each request (normal or error-related) to the file **web_server_log** in the format below. You must also protect the log from multiple threads attempting to access it simultaneously.

[ThreadID#][Request#][fd][Cache HIT/MISS][Request string][bytes/error]
Where:

- **ThreadID#** is an integer from 0 to num_workers -1 indicating the worker-thread ID of the handling thread.

- **Request#** is the total number of requests this specific worker thread has handled so far, including this request
- **fd** is the file descriptor given to you by accept_connection() for this request
- **Cache HIT/MISS** is either ‘HIT’ or ‘MISS’ depending on whether this specific request was found in the cache or not
- **Request** string is the filename buffer filled in by the get request function
- **bytes/error** is either the number of bytes returned by a successful request, or the error string returned by return error if an error occurred.

The server will be configurable in the followings ways:

- **port** number can be specified (you may only use ports 1025 - 65535 by default)
- **path** is the path to your web root location where all the files will be served from
- **num_dispatch** is how many dispatch threads to start up
- **num_workers** is how many worker threads to start up
- **num_prefetch** is how many prefetch threads to start up
- **qlen** is the fixed, bounded length of the request queue
- **mode** can be FCFS (value=0) or CRF (value=1) or SFF (value=2)
- **cache-entries** is the number of entries available in the cache (an alternative way to do this would be to specify the maximum memory used by the cache, but we are just using a limit on the number of entries).

Your server should be run as:

% **web_server_http port path num_dispatch num_workers num_prefetch qlen mode cache_entries**

7 Provided Files and How To Use Them

We have provided many functions which you must use in order to complete this assignment. We have handled all of the networking system calls for you. We have also handled the HTTP protocol parsing for you and we have provided a function that you will use to guess the next request.

Some of the library function calls assumes that the program has “chdir”ed to the web tree root directory. You need to make sure that you chdir to the web tree root somewhere in the beginning.

We have provided a makefile you can use to compile your server. Here is a list of the files we have provided.

1. **makefile** You can use this to compile your program using our object files, or you can make your own. You can study this to see how it compiles our object code along with your server code to produce the correct binary executables.
2. **util.h** This contains a very detailed documentation of each function **that you must**

study and understand before using the functions.

3. **util-http.o** This is the compiled code of the functions described in util.h to be used for the web server. Compile this into your multi-threaded server code and it will produce a fully-functioning web server.

Remember to use -D REENTRANT when compiling your C source files and –lpthread when linking.

8 Simplifying Assumptions

- The maximum number of dispatch threads will be 100
- The maximum number of worker threads will be 100
- The maximum number of prefetcher threads will be 100
- The maximum size of the cache will be 100 entries
- The maximum length of the request queue will be 100 requests
- Any HTTP request for a filename containing two consecutive periods or two consecutive slashes ("..” or “//”) will automatically be detected as a bad request by our compiled code for security reasons.

9 Documentation

Exactly as described in previous specifications in this course, you must provide a README file which describes your program as previously outlined. At the top of your README file and main C source file, please include the following comment:

```
/* csci4061 F2011 Assignment 4
*section: one_digit_number
*login: itlabs_login_name(login used to submit)
*date: mm/dd/yy
*names: Name of each member of the team
*ID: ID of each member of the team
```

10 Grading

25% README, Documentation within code, coding and style very important (indentations, readability of code, use of defined constants rather than numbers)
75% Correctness, error handing, meeting the specifications
• Make sure to pay attention to documentation and coding style. A perfectly working program will not receive full credit if it is undocumented and very difficult to read.
• We will use GCC version 4.2 to compile your code.
• The grading will be done on ITLabs machines only.