ENCE 360 Assignment - 2020

Task

Your task is to write (some parts of) a multi-threaded HTTP multipart downloader program.

There are two parts:
Part 1 – a minimal HTTP client
Part 2 - a concurrent queue using semaphores.

Part 3 – determining chunk sizes and the number of partial downloads required

Part 4 – merging & removing partial content files

A downloader is written in terms of these parts and you will do some analysis and write a short report.

All the programs can be compiled using a Makefile provided simply by going to the base directory where you unpack the assignment and typing '**make**'

Guidelines

- No more than 3 levels of nesting in any function and less than 40 lines of code
- Use minimum amount of code required
- Do not modify any code outside of **http.c, queue.c** and **downloader.c** (or the test programs which you may modify to improve – but they will not be marked)**,** otherwise your submission will fail to compile (and you will receive zero marks)
- All memory allocated during the programs execution should be free()'ed and all resources, files, sockets should be closed before the program finishes. **Check your program with valgrind –leakcheck=all to be sure!**
- Comment code as necessary, excessive commenting is not required but anything not obvious should be documented
- Marks will be awarded for clean code and documentation

Submission

Submit files in the same directory structure as provided, with an extra file called report.pdf in the base directory, with the implementation files queue.c, http.c and downloader.c completed.

Please do NOT MODIFY any header files (.h files) because your files will be compiled using the original .h files and any modifications will be ignored.

Your submission will be automatically marked, so any submission which does not fit the correct template won't receive any marks.

Provided files

src/

downloader.c

A multi-threaded downloader based on your solutions to Parts 1, 2, 3 & 4.

http.c http.h

Http client implementation, Part 1 & Part 3.

queue.c queue.h

A skeleton for the concurrent queue, Part 2.

test/

http_test.c queue_test.c

These are provided to test your implementations for Part 1 and Part 2.

Part 1 – http client (30%) (20%)

Write an implementation of an http client in http.c which performs an HTTP 1.0 query to a website and returns the response string (including header and page content). Note that a server may not respect the range size you specify, so be wary of the webpages you test against. A good one to use is unsplash.com.

Make sure to test your implementation against a selection of binary files, and text files – small files and big. Be very careful using string manipulation functions, e.g. **strcpy** – they will not copy any binary data containing a '\0' – so you will want to use the binary counterparts such as **memcpy**.

For downloading bigger files you will need to think about how to allocate memory as you go – functions such as **realloc** allow you to dynamically resize a buffer without copying the contents each time.

Functions which you will need to implement this include: memset, getaddrinfo, socket, connect, realloc, memcpy, perror and free

Also, possibly useful: **fdopen dup**

DO NOT base your code on this code from the internet – e.g. http://coding.debuntu.org/c-linux socket-programming-tcp-simple-http-client

It uses old style socket programming (all the wrong functions) and will only waste you time and energy, you may get ideas however.

Your lecture notes may be useful, and this is a good reference which you may find useful: http://beej.us/guide/bgnet/output/html/multipage/syscalls.html

Sample HTTP GET request

To retreive the file www.canterbury.ac.nz/postgrad/documents/cribphdmay.doc

```
"GET /page HTTP/1.0\r\n
Host: host\r\n
Range: bytes=0-500\r\n
User-Agent: getter\r\n\r\n"
```

Where host = "www.canterbury.ac.nz" and page = "postgrad/documents/cribphdmay.doc". The *Range* portion is optional, but when specified allows you to retrieve part of a file. See this MDN page for more information.

Example output

Test programs **http_test**, and **http_download** is provided for you to test your code, an example output of the **http_test** is shown below. It is implemented in **test/http_test.c** and is built by the Makefile by default.

```
./http_test www.thomas-bayer.com sqlrest/CUSTOMER/3/
Header:
HTTP/1.1 200 OK
Server: ApacheCoyote/1.1
ContentType: application/xml
Date: Tue, 02 Sep 2014 04:47:16 GMT
Connection: close
ContentLength: 235

Content:
<?xml version="1.0"?><CUSTOMER
xmlns:xlink="http://www.w3.org/1999/xlink">
    <ID>3</ID>
    <FIRSTNAME>Michael</FIRSTNAME>
    <LASTNAME>Clancy</LASTNAME>
    <STREET>542 Upland Pl.</STREET>
    <CITY>San Francisco</CITY>
</CUSTOMER>
```

http_download does a similar thing but instead writes the downloaded file to disk using a filename you give it. A script is provided to test your implementation against those of well known file downloader **wget**

```
./test_download.sh
www.cosc.canterbury.ac.nz/research/reports/PhdTheses/2015/phd_1501
.pdf
downloaded 13025535 bytes from
www.cosc.canterbury.ac.nz/research/reports/PhdTheses/2015/phd_1501 .pdf

Files __template and __output are identical
./test_download.sh
static.canterbury.ac.nz/web/graphics/blacklogo.png

downloaded 9550 bytes from

static.canterbury.ac.nz/web/graphics/blacklogo.png Files
__template and __output are identical
```

Part 2 – concurrent queue (20%)

Write a classic implementation of a concurrent FIFO queue in **queue.c** which allows multiple producers and consumers to communicate in a thread-safe way. Before studying the requirements, it is advised to study the test program **queue_test** for an example of how such a queue is intended to be used.

Hints: Use semaphores (sem_init, sem_wait, sem_post etc.) and mutex(s) for thread synchronization. Use a minimum number of synchronization primitives while still maintaining correctness and maximum performance.

Testing

A test program **queue_test** is provided for you to test your code and illustrate how to use the concurrent queue. Note that it is not a completely comprehensive test – and the test used for marking will be much more stringent on correctness, it may be possible to run the queue_test program yet receive low marks. So you may wish to write your own tests and/or test with the provided **downloader** program.

```
./queue_test
total sum: 1783293664, expected sum: 1783293664
```

(This should complete within two seconds on the lab machines)

Part 3 – chunk sizes & partial-content downloads (5%)

In **http.c,** write an implementation to determine the maximum byte size to request in a partial download. You should use this to determine the number of partial-content downloads your program needs to execute. A server *may not* respect the range size. It is recommended to use a *HEAD* request here since only the headers are desired. The request can be performed similar to a GET request:

```
"HEAD /page HTTP/1.0\r\n
Host: host\r\n
User-Agent: getter\r\n\r\n"
```

Where host = “images.unsplash.com” and page = “photo-1593462131651-9f3d58eecd9e?ixlib=rb-1.2.1&ixid=eyJhcHBfaWQiOjEyMDd9&auto=format&fit=crop&w=1355&q=80”

Part 4 – File Merging & Removal (15%)

In **downloader.c,** two function prototypes have been defined for you. **Merge_files** and **remove_chunk_files**. Implement merge_files first, and check that your output is as expected. When your file merging is working correctly, explore removing the partial-download files.

NOTE: Feel free to discard either of these functions and use a more optimal approach (explain this in your report).

Part 5 – report (40%)

Algorithm analysis

Describe the algorithm found in downloader.c (lines 200-230) step by step.

How is this similar to algorithms found in your notes, which is it most like – and are there any improvements which could be made?

Performance analysis

Provide a small analysis of performance of the downloader program showing how performance changes as the number of worker threads increases. What is the optimal number of worker threads? What are some of the issues when the number of threads is increased?

Run the downloader several times over different inputs (I have provided some urls e.g. test.txt and large.txt) with small files, and large files to download.

Usage (downloading files in test.txt with 12 threads and saving to directory ‘`download`’):

```
./downloader test.txt 12 download
```

How does the file size impact on performance? Are there any parts of your http downloader which can be modified to download files faster?

A pre-compiled version of the downloader exists in ‘**bin/downloader**’ how does your implementation compare? If there’s a large discrepancy when the number of threads increase, it’s likely there’s something wrong with your concurrent queue!

Analyse the approach used for file merging and removal. Are there any improvements that can be made? Why? Is the current method of partially downloading every file optimal? If not, suggest a way to improve performance.

Tip: Answer every question here in your report to maximise your marks. They are all worth something!

Submit this as report.pdf with your submission