Patrick Sheehan
CSCE 313H - MP4
23 April 2014

Design

Getweb.c is a single-threaded file downloader for http. When compiled ('gcc getweb.c'), it takes one command line argument, a URL. This URL should point to a file online that can be downloaded over hypertext transfer protocol. The program will break down this string into the host name, the path to the file within the website, and the name of the file itself. Using this, a GET request is built using the syntax, GET <url> HTTP/1.1 <newline> Host: <hostname>. A single unix socket to the host is then created, and the GET request is sent. Next, the recv function is used on the socket to accept small chunks of data. Each chunk is stored in a buffer, written to the user's new file, and the process repeats until the file is fully downloaded.

Mgetweb.c uses multiple threads (defined in code by THREAD_COUNT) to download a file from http in a manner similar to getweb.c. The URL is supplied as a command line argument and is broken down the same way. However, in this program, a HEAD request is built first. The reason for this is to learn some information about the file, namely its size, before downloading the entire thing. Retrieving the HEAD first allows us to see if the request for the file is valid, how many bytes are in the file, and if the file can accept byte ranges. This gives us the option to send a more specific GET request. For this project, two different ways of downloading the file were tried. First, I will explain the method that I abandoned, then the alternative that turned out to be successful.

1. Using the number of bytes obtained by the header, we can assign a range of bytes that each thread is responsible for downloading (bytes_in_file / THREAD_COUNT). The GET request can then be formed in the same way as above, but with the addition of 'Range: bytes=<start_byte>-<end_byte>'. I originally tried letting each thread have it's own unique GET request with a range, storing it to a buffer, then sending a pointer to that buffer to the calling thread on the pthread_join function. This was unsuccessful, possibly due to synchronization issues, or memory storage/leakage.
2. The successful method was sending a GET request for the whole file (as in getweb.c) through a data socket, then allowing each thread to receive small chunks at a time. The threads are concurrently listening to the socket, and each time data is received, the small amount of data in the buffer is written to the global FILE pointer. Surprisingly, mutex locks were not needed, so the compiler might be treating the section of code as atomic. Once there is no more receiving to be done, the master thread joins on all of its children and closes the brand new downloaded file for the user to enjoy, at a faster rate.

Results

The quantitative analysis data can be found in the 'Timing Analysis' spreadsheet.

Findings

Surprisingly, the timing among different numbers of threads were not very distinguishable amongst themselves, but they were significantly faster than the single threaded version for large files.

Thoughts

This project gave me a much better understanding of many aspects of low-level system and network programming. It also begs new questions for how to decide the most efficient way of the tasks that we as users take for granted.

Sources:

cboard.cprogramming.com
lecture slides
pubs.opengroup.org
http://shoe.bocks.com/net, beej.us guides,
linux.die.net/man
cplusplus.com
linux.about.com

Here were the files used for download testing:

835kb - http://courses.cse.tamu.edu/guofei/csce313/1_Introduction%20to%20OS.pdf
10mb - http://vhost2.hansenet.de/10_mb_file.bin
100mb - http://lg.denver.fdcservers.net/100MBtest.zip
500mb - http://speedtest.clt.carohosting.com/500MB.dtf
1gb - http://speedtest.reliableservers.com/1GBtest.bin