CSCE 463/612: Networks and Distributed Processing

Homework 1 Part 3 (50 pts)

Due date: 2/15/22

1. Problem Description

We are finally ready to multi-thread this program and achieve significantly faster download rates. Due to the high volume of outbound connections, your home ISP (e.g., Suddenlink, campus dorms) will probably block this traffic and/or take your Internet link down. Do not be alarmed, this condition is usually temporary, but it should remind you to run the experiments over VPN. The program may also generate high rates of DNS queries against your local server, which may be construed as malicious. In such cases, it is advisable to run your own version of BIND on localhost.

1.1. Code (25 points)

The command-line format remains the same as in Part 2, but allows more threads:

```
hw1.exe 3500 URL-input.txt
```

To achieve proper load-balancing, you need to create a shared queue of pending URLs, which will be drained by the crawling threads using an unbounded producer-consumer from CSCE 313. The general algorithm follows this outline:

```
int _tmain(int argc, _TCHAR* argv[])
{
	// parse command line args
	// initialize shared data structures & parameters sent to threads

	// read file and populate shared queue
	// start stats thread
	// start N crawling threads

	// wait for N crawling threads to finish
	// signal stats thread to quit; wait for it to terminate
	// cleanup
}
```

The output should be printed by the stats thread every 2 seconds:

```
[  6]  500 Q 992142 E    7862 H   1790 D   1776 I  1264 R   544 C   190 L    5K
      *** crawling 87.5 pps @ 12.3 Mbps
```

The first column is the elapsed time in seconds (to achieve 3-character alignment, use `%3d` in `printf`). The next column shows the number of active threads (i.e., those that are still running). As the program nears shutdown, you will see this number slowly decay towards zero. The remaining columns are labeled with a single letter whose meaning is given below:

```
Q: current size of the pending queue
E: number of extracted URLs from the queue
H: number of URLs that have passed host uniqueness
D: number of successful DNS lookups
```

```
I: number of URLs that have passed IP uniqueness
R: number of URLs that have passed robots checks
C: number of successfully crawled URLs (those with a valid HTTP code)
L: total links found
```

Note that proper alignment of columns is required. You will need six character positions for Q, seven for E, six for (H, D), five for (I, R, C), and four for L. The second line of the example prints the crawling speed in pages per second (pps) and the download rate in Mbps, computed over the period since the last report. You will need to determine the number of pages/bytes downloaded and the elapsed time between wakeups in the stats thread, then divide the two. For a more accurate bandwidth usage, you should combine both robots and page bytes; however, the crawling speed only refers to non-robot pages.

At the end, the following summary should be printed (see the traces in Section 1.6 for the details on how each of these numbers is produced from earlier printouts):

```
Extracted 1000004 URLs @ 9666/s
Looked up 139300 DNS names @ 1346/s
Attempted 95460 robots @ 923/s
Crawled 59904 pages @ 579/s (1651.63 MB)
Parsed 3256521 links @ 31476/s
HTTP codes: 2xx = 47185, 3xx = 5826, 4xx = 6691, 5xx = 202, other = 0
```

1.2. Report (25 points)

The report should address the following questions based on the links in `URL-input-1M.txt`:

1. (5 pts) Briefly explain your code architecture and lessons learned. Using Part 3, show a complete trace with 1M input URLs.

2. (5 pts) Across all pages that came back with a 2xx code, calculate the average number of HTML links (i.e., out-neighbors) found by the parser. Estimate the size of Google's webgraph (in terms of edges and bytes it occupies on disk) assuming they crawl 1T (trillion) pages. A webgraph here would store each crawled node x and its out-neighbors (y_1, y_2, …) using adjacency lists, where URLs are represented by 64-bit hashes.

3. (5 pts) Determine the average page size in bytes (across all HTTP codes). Estimate the bandwidth (in Gbps) needed for Bing to crawl 10B pages a day.

4. (5 pts) What is the probability that a link in the input file contains a unique host? What is the probability that a unique host has a valid DNS record? What percentage of contacted sites had a 4xx robots file?

5. (5 pts) How many of the crawled 2xx pages contain a hyperlink to our domain `tamu.edu`? How many of them originate from outside of TAMU? Explain how you obtained this information. Examples of suitable links:

   ```
   irl.cs.tamu.edu/
   afcerc.tamu.edu/index.html
   tamu.edu/
   www.cse.tamu.edu/people
   ```

Examples of false-positives:

```
tamu.edu.cn/
www.x.com/tamu.edu/
```

1.3. Parser

The parser is not multi-threaded safe and thus should not be called from multiple threads. It maintains an internal buffer of produced links that gets overwritten in each call. One option is to enclose all parser-related functionality in a mutex; however, this prevents concurrent parsing of pages and hurts performance. For maximum speed, the best approach is to create a separate instance of the parser inside each thread. This prevents corruption of the shared buffer and avoids the need for synchronization. It is not advisable to create/delete the parser for each URL; instead, create it once when the thread starts and keep using it for all subsequent links.

1.4. Synchronization and Threads

It is a good idea to learn Windows threads and synchronization primitives by running and dissecting the sample project on the course website. As long as you remember the main concepts from CSCE 313, most of the APIs are pretty self-explanatory and have good coverage on MSDN. The main synchronization algorithm you will be using is called *producer-consumer*. In fact, our problem is slightly simpler and can be solved using the following:

```
Producer ()           // called by _tmain()
{
        // produce items into the queue
        for (i = 0; i < N; i++)
                Q.push (host [i]);
}

Consumer ()           // crawling thread
{
        while (true)
        {
                mutex.Lock ();
                if (Q.size() == 0)            // finished crawling?
                {
                        mutex.Unlock();
                        break;
                }
                x = Q.front (); Q.pop();
                mutex.Unlock ();
                // crawl x
        }
}
```

For mutexes, there is a user-mode pair of functions `EnterCriticalSection` and `LeaveCriticalSection` that operate on objects of type `CRITICAL_SECTION`. Note that you must call `InitializeCriticalSection` before using them. You can also use kernel mutexes created via `CreateMutex`, but they are much slower.

To update the stats, you can use a critical section, but it is often faster to directly use interlocked operations, each mapping to a single CPU instruction. You may find `InterlockedIncrement` and `InterlockedAdd` useful.

After emptying the input queue, most of the threads will quit successfully, but some will hang for an extra 20-30 seconds, which will be caused by `connect()` and `select()` waiting on timeout. There is no good way to reduce the shutdown delay unless you employ overlapped or non-blocking sockets (i.e., using `WSA_FLAG_OVERLAPPED` in `WSASocket` or `FIONBIO` in `ioctlsocket`). These are not required, but can be explored for an additional level of control over your program.

Quit notification can be accomplished with a manual event. See `CreateEvent` and `SetEvent`. For example, the stats thread boils down to a simple loop waiting for this event:

```
DWORD WINAPI StatsRun(LPVOID lpPara)
{
	Parameters *p = (Parameters*) lpPara;	// shared parameters
	while (WaitForSingleObject (p->eventQuit, 2000) == WAIT_TIMEOUT)
	{
		// print
	}
}
```

Note that the `Parameters` structure can accommodate other shared state:

```
DWORD WINAPI CrawlerRun(LPVOID lpPara)
{
	Parameters *p = (Parameters*) lpPara;	// shared parameters
	while (true)
	{
		EnterCriticalSection (&p->cs);
		if (p->Q.size () == 0)
			...
		LeaveCriticalSection (&p->cs);
	}
}
```

1.5. Extra Credit (20 pts)

To receive extra credit, you must be able to process HTTP 1.1 responses that are chunked. This will be checked using Part-1 functionality of the homework (i.e., one command-line argument). Please specify in the report that your program can do HTTP 1.1 downloads. This will be checked with the final part of the homework.

Chunking is indicated by the "Transfer-Encoding" field in the response:

```
GET / HTTP/1.1
Host: tamu.edu
User-agent: myTAMUcrawler/1.0
Connection: close

HTTP/1.1 200 OK\r\n
Connection: close\r\n
Date: Thu, 1 Sep 2006 12:00:15\r\n
Server: Apache/1.3.0\r\n
Content-type: text/html\r\n
Transfer-Encoding: chunked\r\n
\r\n
2A0\r\n
<html><head><meta http-equiv="Content-Language" content="en-us">...
0\r\n
```

In these cases, the data following the header is split into blocks, each of which is preceded by a hex number that specifies its size. As there may be many such segments, the last one has size 0.

For all 2xx pages, print an extra line indicating the body length (i.e., page size without the HTTP header) before and after dechunking.

```
URL: http://jigsaw.w3.org/HTTP/ChunkedScript
        Parsing URL... host jigsaw.w3.org, port 80, request /HTTP/ChunkedScript
        Doing DNS... done in 0 ms, found 128.30.52.21
      * Connecting on page... done in 46 ms
        Loading... done in 203 ms with 72563 bytes
        Verifying header... status code 200
        Dechunking... body size was 72268, now 72200
      + Parsing page... done in 0 ms with 0 links

----------------------------------------
HTTP/1.1 200 OK
cache-control: max-age=0
date: Tue, 18 Aug 2020 21:04:59 GMT
transfer-encoding: chunked
content-type: text/plain
etag: "1j3k6u8:tikt981g"
expires: Tue, 18 Aug 2020 21:04:58 GMT
last-modified: Mon, 18 Mar 2002 14:28:02 GMT
server: Jigsaw/2.3.0-beta2
connection: close
```

Note that dechunking *in place* is the preferred approach. This can be done using repeated `memcpy` operations within the buffer, i.e., shifting chunks up to eliminate the gaps. Also, if you plan to use string functions to find the transfer-encoding field, make sure to NULL-terminate the buffer. Otherwise, `strstr` may escape the buffer and cause a crash. Finally, since HTTP fields are case-insensitive, you should use `StrStrI` in your search.

1.6. Traces

The results below were collected in 2015, which may differ from the outcome today. If you would like a more up-to-date trace, you can post a question on Piazza. The first example uses URL-input-100.txt and 10 threads:

```
Opened URL-input-100.txt with size 6003
[  2]   10 Q     41 E      59 H     55 D     55 I    50 R     8 C     0 L    0K
      *** crawling 0.0 pps @ 0.1 Mbps
[  4]   10 Q     16 E      84 H     75 D     75 I    66 R    10 C     5 L    0K
      *** crawling 2.5 pps @ 0.4 Mbps
[  6]    4 Q      0 E     100 H     84 D     84 I    74 R    12 C     7 L    1K
      *** crawling 1.0 pps @ 0.4 Mbps

Extracted 100 URLs @ 13/s
Looked up 84 DNS names @ 11/s
Attempted 74 robots @ 9/s
Crawled 11 pages @ 1/s (0.23 MB)
Parsed 543 links @ 70/s
HTTP codes: 2xx = 7, 3xx = 4, 4xx = 0, 5xx = 0, other = 0
```

The next run was obtained using 5000 threads and URL-input-1M.txt:

```
Opened URL-input-1M.txt with size 66152005
[  2] 5000 Q 950541 E   49462 H   6489 D   6448 I  5014 R  2207 C     5 L    0K
      *** crawling 2.5 pps @ 4.2 Mbps
[  4] 5000 Q 932781 E   67223 H  10456 D  10387 I  8008 R  4568 C   123 L    5K
      *** crawling 58.6 pps @ 12.8 Mbps
[  6] 5000 Q 902274 E   97728 H  14567 D  14467 I 11532 R  6556 C  2578 L   99K
      *** crawling 1217.6 pps @ 207.8 Mbps
[  8] 5000 Q 877451 E  122553 H  18130 D  18010 I 14598 R  8676 C  4386 L  215K
```

```
      *** crawling 894.6 pps @ 223.3 Mbps
[ 10] 5000 Q 854240 E  145764 H  21680 D  21526 I 17691 R 10648 C  6522 L  343K
      *** crawling 1060.1 pps @ 236.1 Mbps
[ 12] 5000 Q 830930 E  169074 H  25105 D  24924 I 20626 R 12641 C  8443 L  457K
      *** crawling 952.8 pps @ 227.3 Mbps
[ 14] 5000 Q 803317 E  196686 H  28467 D  28240 I 23486 R 14580 C 10375 L  561K
      *** crawling 958.3 pps @ 193.2 Mbps
[ 16] 5000 Q 778447 E  221557 H  32003 D  31728 I 26444 R 16463 C 12227 L  665K
      *** crawling 918.0 pps @ 198.9 Mbps
[ 18] 5000 Q 754649 E  245355 H  35431 D  35107 I 29280 R 18386 C 14045 L  766K
      *** crawling 900.6 pps @ 207.0 Mbps

...

[ 76] 5000 Q  44366 E  955638 H 133113 D 131807 I 92391 R 60734 C 55962 L 2998K
      *** crawling 502.7 pps @ 93.7 Mbps
[ 78] 5000 Q  17924 E  982080 H 136261 D 134919 I 93926 R 61695 C 56975 L 3072K
      *** crawling 500.5 pps @ 132.7 Mbps
[ 80] 3223 Q      0 E 1000004 H 139200 D 137818 I 95413 R 62755 C 57872 L 3112K
      *** crawling 445.1 pps @ 89.7 Mbps
[ 83] 2111 Q      0 E 1000004 H 139279 D 137888 I 95457 R 62855 C 58766 L 3152K
      *** crawling 444.3 pps @ 76.8 Mbps
[ 85] 1185 Q      0 E 1000004 H 139288 D 137891 I 95459 R 62864 C 59529 L 3193K
      *** crawling 378.2 pps @ 76.7 Mbps
[ 87]  543 Q      0 E 1000004 H 139292 D 137892 I 95460 R 62867 C 59800 L 3246K
      *** crawling 132.6 pps @ 89.6 Mbps
[ 89]  410 Q      0 E 1000004 H 139296 D 137892 I 95460 R 62867 C 59857 L 3252K
      *** crawling 28.3 pps @ 17.2 Mbps
[ 91]  311 Q      0 E 1000004 H 139296 D 137892 I 95460 R 62868 C 59890 L 3255K
      *** crawling 15.9 pps @ 6.5 Mbps
[ 93]  233 Q      0 E 1000004 H 139300 D 137892 I 95460 R 62868 C 59900 L 3256K
      *** crawling 5.0 pps @ 8.7 Mbps
[ 95]  170 Q      0 E 1000004 H 139300 D 137892 I 95460 R 62868 C 59901 L 3256K
      *** crawling 0.5 pps @ 2.9 Mbps
[ 97]  123 Q      0 E 1000004 H 139300 D 137892 I 95460 R 62868 C 59902 L 3256K
      *** crawling 0.5 pps @ 0.3 Mbps
[ 99]   79 Q      0 E 1000004 H 139300 D 137892 I 95460 R 62868 C 59902 L 3256K
      *** crawling 0.0 pps @ 0.3 Mbps
[101]   22 Q      0 E 1000004 H 139300 D 137892 I 95460 R 62868 C 59904 L 3257K
      *** crawling 1.0 pps @ 1.0 Mbps
[103]    3 Q      0 E 1000004 H 139300 D 137892 I 95460 R 62868 C 59904 L 3257K
      *** crawling 0.0 pps @ 0.0 Mbps

Extracted 1000004 URLs @ 9666/s
Looked up 139300 DNS names @ 1346/s
Attempted 95460 robots @ 923/s
Crawled 59904 pages @ 579/s (1651.63 MB)
Parsed 3256521 links @ 31476/s
HTTP codes: 2xx = 47185, 3xx = 5826, 4xx = 6691, 5xx = 202, other = 0
```

For additional testing, there is another file on the course website (i.e., URL-input-1M-2019.txt), which contains mostly unique hosts. It thus takes significantly longer to run, but provides a much higher fraction of successful downloads.

463/612 Homework 1 Grade Sheet (Part 3)

Name: _____

Function	Points	Break down	Item	Deduction
Running output	12	1	Printouts not every 2 seconds	
		1	Incorrect active threads	
		1	Incorrect Q	
		1	Incorrect E	
		1	Incorrect H	
		1	Incorrect D	
		1	Incorrect I	
		1	Incorrect R	
		1	Incorrect C	
		1	Incorrect L	
		1	Incorrect pps	
		1	Incorrect Mbps	
Summary	6	1	Incorrect URL processing rate	
		1	Incorrect DNS rate	
		1	Incorrect robots rate	
		1	Incorrect crawled rate/totals	
		1	Incorrect parser speed	
		1	Incorrect HTTP breakdown	
Code	6	1	<< 20 Mbps w/500 threads	
		1	>> 200 MB RAM w/500 threads	
		2	Deadlocks on exit	
		1	Issues with the file reader	
		1	Improper stats thread	
Other	1	1	Missing files for compilation	
Report	25	5	Lessons learned and trace	
		5	Google graph-size analysis	
		5	Yahoo bandwidth analysis	
		5	Probability analysis	
		5	In-degree of tamu.edu	

Additional deductions are possible for memory leaks and crashing.

Total points: _____