CSCE 463/612: Networks and Distributed Processing

Homework 1 (100 pts)

Due dates: 1/23/18 (part 1), 1/30/18 (part 2), 2/13/18 (part 3)

1. Purpose

This assignment builds an understanding of text-based application-layer protocols, multi-threading, system APIs, and Windows sockets.

2. Description (Part 1)

Using Visual C++, your goal is to create a simple web client that accepts URLs and then crawls them to display basic server/page statistics.

2.1. Code (25 pts)

Your program must accept a single command-line argument with a target URL. If the argument is missing or there are too many of them, print usage information and quit. You should be able to handle all combinations of URLs that fall under

```
scheme://host[:port][/path][?query][#fragment]
```

Examples:

```
hw1.exe http://tamu.edu                                        [no path, no port, no query]
hw1.exe http://www.tamu.edu:80                                 [no path, port]
hw1.exe http://128.194.135.72:80/courses/index.asp#location [IP, port, path, fragment]
hw1.exe http://165.91.22.70/                                   [IP, path]
hw1.exe http://s6.irl.cs.tamu.edu/IRL140M.htm                  [140MB binary file]
hw1.exe http://s6.irl.cs.tamu.edu/IRL96M.htm                   [96MB HTML file]
hw1.exe http://csnet.cs.tamu.edu:443?addrbook.php              [no path, query, non-HTTP response]
hw1.exe http://smtp-relay.tamu.edu:465/index.html              [path, port, recv error]
hw1.exe http://ftp.gnu.org:21/                                 [recv timeout]
hw1.exe http://s22.irl.cs.tamu.edu:990/view?test=1             [path, port, query, no DNS]
hw1.exe http://128.194.135.11?viewcart.php/                    [no path, query, connect timeout]
```

Note that if the path is omitted, you must use the root directory / in its place. If the URL passes basic checks (i.e., correct scheme, valid port number), you should attempt to obtain the page via HTTP 1.0. Note that HTTP 1.1 allows the server to chunk the transfer, which is harder to decode (see the extra-credit section at the end). It is therefore important to request the version of HTTP that you can process.

If you manage to connect and receive a valid reply (HTTP 2xx), parse the HTML result and display the required information about your download (see below); otherwise, the program should legibly report the type of error encountered and terminate gracefully, *even if the remote host is hanging or not responding*. Note that your code must be able to handle pages of arbitrary length by dynamically expanding the buffer provided to `recv()`.

The following two examples show the required behavior during successful downloads:

```
URL: http://www.dmoz.org
        Parsing URL... host www.dmoz.org, port 80, request /
```

```
        Doing DNS... done in 655 ms, found 149.174.98.43
      * Connecting on page... done in 47 ms
        Loading... done in 141 ms with 17839 bytes
        Verifying header... status code 200
      + Parsing page... done in 0 ms with 102 links

---------------------------------------
HTTP/1.1 200 OK
Date: Sun, 18 Jan 2015 01:52:21 GMT
Server: Apache
Set-Cookie: JSESSIONID=2C016DC9413B184589C279886C0BA9D0; Path=/
Content-Length: 17621
Connection: close
Content-Type: text/html;charset=UTF-8
```

and

```
URL: http://128.194.135.72
        Parsing URL... host 128.194.135.72, port 80, request /
        Doing DNS... done in 0 ms, found 128.194.135.72
      * Connecting on page... done in 0 ms
        Loading... done in 78 ms with 6957 bytes
        Verifying header... status code 200
      + Parsing page... done in 0 ms with 10 links

---------------------------------------
HTTP/1.1 200 OK
Cache-Control: private
Content-Length: 6633
Content-Type: text/html
Server: Microsoft-IIS/7.0
Set-Cookie: ASPSESSIONIDAASDCQDS=FIMPNKHBEFLNGPCOGOOPPBHI; path=/
X-Powered-By: ASP.NET
MicrosoftOfficeWebServer: 5.0_Pub
MS-Author-Via: MS-FP/4.0
Date: Sun, 18 Jan 2015 01:54:01 GMT
Connection: close
```

Note that one-tab indentation after the first line, an asterisk for the connection phase, a plus for the parsing phase, and timing of each networking step are required. The printout following a horizontal line contains only the HTTP header.

If you manage to receive the page, but the status code is not 2xx, skip the HTML parser, but print everything else:

```
URL: http://www.yahoo.com
        Parsing URL... host www.yahoo.com, port 80, request /
        Doing DNS... done in 484 ms, found 98.138.253.109
      * Connecting on page... done in 31 ms
        Loading... done in 109 ms with 1746 bytes
        Verifying header... status code 301

---------------------------------------
HTTP/1.0 301 Redirect
Date: Sun, 18 Jan 2015 01:39:06 GMT
Via: http/1.1 ir24.fp.ne1.yahoo.com (ApacheTrafficServer)
Server: ATS
Location: https://www.yahoo.com/
Content-Type: text/html
Content-Language: en
Cache-Control: no-store, no-cache
Connection: keep-alive
Content-Length: 1450
```

If you are unable to download the page, stop at the last attempted step and display the failure condition (for network errors, provide the corresponding `WSAGetLastError()` result):

```
URL: http://csnet.cs.tamu.edu:443?addrbook.php
        Parsing URL... host csnet.cs.tamu.edu, port 443, request /?addrbook.php
        Doing DNS... done in 5 ms, found 128.194.138.14
      * Connecting on page... done in 2 ms
        Loading... failed with non-HTTP header
```

```
URL: http://smtp-relay.tamu.edu:465/index.html
        Parsing URL... host smtp-relay.tamu.edu, port 465, request /index.html
        Doing DNS... done in 4 ms, found 165.91.22.120
      * Connecting on page... done in 2 ms
        Loading... failed with 10054 on recv
```

```
URL: http://128.194.135.11?viewcart.php/
        Parsing URL... host 128.194.135.11, port 80, request /?viewcart.php/
        Doing DNS... done in 4 ms, found 128.194.135.11
      * Connecting on page... failed with 10060
```

```
URL: http://s22.irl.cs.tamu.edu:990/view?test=1
        Parsing URL... host s22.irl.cs.tamu.edu, port 990, request /view?test=1
        Doing DNS... failed with 11001
```

```
URL: http://xyz.com:/
        Parsing URL... failed with invalid port
```

```
URL: http://xyz.com:0
        Parsing URL... failed with invalid port
```

```
URL: https://yahoo.com
        Parsing URL... failed with invalid scheme
```

2.2. General Guidelines

Efficient coding and well-structured programming is expected. You may lose points for copy-pasting the same function (with minor changes) over and over again, for writing poorly designed or convoluted code, *not checking for errors in every API you call*, and allowing buffer overflows, access violations, debug-assertion failures, heap corruption, synchronization bugs, memory leaks, or any conditions that lead to a crash. Furthermore, your program must be robust against unexpected responses from the Internet and deadlocks.

Basic operation of Winsock is covered in class, with supporting examples provided in the sample homework project on the course website. Additional caveats are discussed next.

2.3. Receive Loop

Reading from sockets is accomplished using this general algorithm that resizes the buffer as needed:

```
class Socket {
       SOCKET  sock;                // socket handle
       char    *buf;                // current buffer
       int     allocatedSize;       // bytes allocated for buf
       int     curPos;              // current position in buffer
       ...                          // extra stuff as needed
};
```

```
Socket::Socket ()
{
	// allocate this buffer once, then possibly reuse for multiple connections in Part 3
	buf = new char [INITIAL_BUF_SIZE];		// start with 8 KB
}

bool Socket::Read (void)
{
	// set timeout to 10 seconds
	while (true)
	{
		// wait to see if socket has any data (see MSDN)
		if ((ret = select (0, &fd, ..., timeout)) > 0)
		{
			// new data available; now read the next segment
			int bytes = recv (sock, buf + curPos, allocatedSize – curPos, ...);

			if (errors)
				// print WSAGetLastError()
				break;

			if (connection closed)
				// NULL-terminate buffer
				return true;		// normal completion

			curPos += bytes;		// adjust where the next recv goes

			if (allocatedSize – curPos < THRESHOLD)
				// resize buffer; besides STL, you can use
				// realloc(), HeapReAlloc(), or memcpy the buffer
				// into a bigger array
		}
		else if (timeout)
			// report timeout
			break;
		else
			// print WSAGetLastError()
			break;
	}

	return false;
}
```

The above fragment checks the socket to see if there is any data before attempting a receive. Without this, you may experience deadlocks inside `recv()` when the remote host neither provides any data nor closes the connection. Since `select()` modifies the parameters you pass to it, you must reinsert `sock` into `fd_set` each time you call `select()`. This is accomplished with macros FD_ZERO and FD_SET. For more details, see

http://msdn.microsoft.com/en-us/library/ms740141(VS.85).aspx

A cleaner alternative to traditional Unix-style `select()` is `WSAEventSelect()` or the IOCP framework. The former lets you register an event that gets signaled when the socket has data in it. This allows your code to wait for multiple events and implement simple timeout-based socket disconnection. The latter is much more complicated and should be attempted only if the rest of the homework appears too simple:

http://msdn.microsoft.com/en-us/library/windows/desktop/aa365198(v=vs.85).aspx

2.4. Required HTTP Fields

The format of GET requests was shown in class. At minimum, you need to transmit the request line and the host string with the name of the server. For example:

```
GET /some/page/index.php?status=15 HTTP/1.0
Host: tamu.edu
```

However, this request may keep the connection open for some non-compliant servers, which makes it difficult to detect the end of transfer. You therefore may want to explicitly request that the server close the connection:

```
GET /some/page/index.php?status=15 HTTP/1.0
Host: tamu.edu
Connection: close
```

It is also common courtesy to specify your user-agent to keep webmasters aware of visiting browsers and robots. In fact, some websites (e.g., akamai.com) refuse to provide a response unless the user-agent is present in the request header:

```
GET /some/page/index.php?status=15 HTTP/1.0
User-agent: myTAMUcrawler/1.0
Host: tamu.edu
Connection: close
```

You should invent your own string in the format of `crawlerName/x.y`, where x.y can evolve from 1.1 to 1.3 as you progress through the parts of this homework.

2.5. Parser

The parser `.lib` and `.h` files can be downloaded from the course website. There are four different libraries that cover all possible combinations of Debug/Release/win32/x64, where the proper file is automatically determined by `HTMLParserBase.h`. It is recommended that you experiment with the test project and copy the library files into the same directory as your source code.

2.6. Helpful Functions, Tools, and Commands

You can use C-string functions `strchr` and `strstr` to quickly find substrings in a buffer. Comparison is usually performed using `strcmp/stricmp` or `strncmp/strnicmp`. It is recommended to use `printf` as it greatly reduces the amount of typing in this homework compared to `cout`. You can also use `sprintf` to assemble the various parts of a request.

Oftentimes, it is convenient to declare a fixed-size buffer that is large enough to accept even the longest link. To help with this, `HTMLParserBase.h` defines two constants `MAX_HOST_LEN` and `MAX_REQUEST_LEN` that upper-bound the examples we consider valid for this homework. If the user provides a string that violates either bound, you should reject it. An explicit check is required, especially in Part 3 where some of the crawled URLs are known to violate the maximum allowed host length.

Usage of `gethostbyname` for DNS lookups, printout of IPs via `inet_ntoa`, and connection to a server are provided in the sample solution.

For debugging responses, use an HTTP sniffer, e.g., http://testuri.org/sniffer, or various Firefox add-ons. If you need to see the contents of your outgoing packets, use http://www.wireshark.org/. For information about your network configuration, run `ipconfig` at the command prompt (to see the DNS servers, use `ipconfig /all`). To manually perform DNS lookups, try `nslookup host` or `nslookup IP`.

3. Description (Part 2)

You will now expand into downloading multiple URLs from an input file using a single thread. To ensure politeness, you will hit only unique IPs and check the existence of `robots.txt`. Robot exclusion is an informal standard that allows webmasters to specify which directories/files on the server are prohibited from download by non-human agents. See http://www.robotstxt.org/ and https://en.wikipedia.org/wiki/Robots_exclusion_standard for more details. To avoid hanging the code on slow downloads, you will also have to abort all pages that take longer than 10 seconds[1] or occupy more than 2 MB (for robots, this limit is 16 KB).

3.1. Code (25 pts)

The program must now accept either one or two arguments. In the former case, it implements the previous functionality; in the latter case, the first argument indicates the number of threads to run and the second one the input file:

```
hw1.exe 1 URL-input.txt
```

If the number of threads does not equal one, you should reject the parameters and report usage information to the user. Similarly, if the file does not exist or cannot be successfully read, the program should complain and quit. Assuming these checks pass, you should load the file into RAM and split it into individual URLs (one line per URL). You can use `fopen`, `fgets`, `fclose` (or their `ifstream` equivalents) to scan the file one-line-at-a-time. A faster approach is load the entire input into some buffer and then separately determine where each line ends. Use C-style `fread` or an even-faster `ReadFile` for this purpose (the sample HTML-parser project shows usage of `ReadFile`). In the former case, note that the file must be opened in binary mode (e.g., using "rb" in `fopen`) to avoid unnecessary translation that may corrupt the URLs.

To avoid redundant DNS lookups, make sure that only unique hosts make it to `gethostbyname`. Combining this with an earlier discussion of politeness and robots leads to the following logic:

Parse URL → Check host is unique → DNS lookup → Check IP is unique → Request robots → Check HTTP code → Request page → Check HTTP code → Parse page

Note that robot existence should be verified using a HEAD request. This ensures that you receive only the header rather than an entire file. Codes 4xx indicate that the robot file does not exists

[1] Your previous usage of `select` constrained each `recv` to 10 seconds, but allowed unbounded delays across the page. In these cases, a website feeding one byte every 9 seconds could drag forever.

and the website allows unrestricted crawling. Any other code should be interpreted as preventing further contact with that host. Your printouts should begin with indication that you read the file and its size, followed by the following trace:

```
Opened URL-input.txt with size 66152005
URL: http://www.symantec.com/verisign/ssl-certificates
        Parsing URL... host www.symantec.com, port 80
        Checking host uniqueness... passed
        Doing DNS... done in 139 ms, found 104.69.239.70
        Checking IP uniqueness... passed
        Connecting on robots... done in 5 ms
        Loading... done in 57 ms with 213 bytes
        Verifying header... status code 200
URL: http://www.weatherline.net/
        Parsing URL... host www.weatherline.net, port 80
        Checking host uniqueness... passed
        Doing DNS... done in 70 ms, found 216.139.219.73
        Checking IP uniqueness... passed
        Connecting on robots... done in 11 ms
        Loading... done in 61 ms with 179 bytes
        Verifying header... status code 404
      * Connecting on page... done in 3020 ms
        Loading... done in 87 ms with 10177 bytes
        Verifying header... status code 200
      + Parsing page... done in 0 ms with 16 links
URL: http://abonnement.lesechos.fr/faq/
        Parsing URL... host abonnement.lesechos.fr, port 80
        Checking host uniqueness... passed
        Doing DNS... done in 1 ms, found 212.95.72.31
        Checking IP uniqueness... passed
        Connecting on robots... done in 138 ms
        Loading... done in 484 ms with 469 bytes
        Verifying header... status code 404
      * Connecting on page... done in 4335 ms
        Loading... done in 899 ms with 57273 bytes
        Verifying header... status code 200
      + Parsing page... done in 1 ms with 63 links
```

Note that you no longer need to print the request after the port. Uniqueness-verification steps and the robot phase are new and highlighted in bold. If you already have a function that connects to a server, downloads a given URL, and verifies the HTTP header, you can simply call it twice to produce both robots and page-related statistics. The function needs to accept additional parameters that specify a) the HTTP method (i.e., HEAD or GET); b) valid HTTP codes (i.e., 2xx for pages, 4xx for robots); c) maximum download size (i.e., 2 MB for pages, 16 KB for robots); and d) presence of an asterisk in the output. If any of the steps fails, you should drop the current URL and move on to the next:

```
URL: http://allafrica.com/stories/201501021178.html
        Parsing URL... host allafrica.com, port 80
        Checking host uniqueness... failed
URL: http://architectureandmorality.blogspot.com/
        Parsing URL... host architectureandmorality.blogspot.com, port 80
        Checking host uniqueness... passed
        Doing DNS... done in 19 ms, found 216.58.218.193
        Checking IP uniqueness... failed
URL: http://aviation.blogactiv.eu/
        Parsing URL... host aviation.blogactiv.eu, port 80
        Checking host uniqueness... passed
        Doing DNS... done in 218 ms, found 178.33.84.148
        Checking IP uniqueness... passed
        Connecting on robots... done in 9118 ms
        Loading... failed with 10060 on recv
URL: http://zjk.focus.cn/
```

```
        Parsing URL... host zjk.focus.cn, port 80
        Checking host uniqueness... passed
        Doing DNS... done in 1135 ms, found 101.227.172.52
        Checking IP uniqueness... passed
        Connecting on robots... done in 367 ms
        Loading... done in 767 ms with 140 bytes
        Verifying header... status code 403
      * Connecting on page... done in 3376 ms
        Loading... failed with slow download
URL: http://azlist.about.com/a.htm
        Parsing URL... host azlist.about.com, port 80
        Checking host uniqueness... passed
        Doing DNS... done in 81 ms, found 207.126.123.20
        Checking IP uniqueness... passed
        Connecting on robots... done in 5 ms
        Loading... failed with exceeding max
URL: http://apoyanocastigues.mx/
        Parsing URL... host apoyanocastigues.mx, port 80
        Checking host uniqueness... passed
        Doing DNS... done in 57 ms, found 23.23.109.126
        Checking IP uniqueness... passed
        Connecting on robots... done in 49 ms
        Loading... done in 2131 ms with 176 bytes
        Verifying header... status code 404
      * Connecting on page... done in 3051 ms
        Loading... failed with exceeding max
URL: http://ba.voanews.com/media/video/2563280.html
        Parsing URL... host ba.voanews.com, port 80
        Checking host uniqueness... passed
        Doing DNS... done in 11 ms, found 128.194.178.217
        Checking IP uniqueness... passed
        Connecting on robots... done in 2 ms
        Loading... done in 490 ms with 2436 bytes
        Verifying header... status code 404
      * Connecting on page... done in 3001 ms
        Loading... done in 50 ms with 2850 bytes
        Verifying header... status code 408
```

In the last example, the downloaded page does not result in success codes `2xx`, which explains why parsing was not performed. As the text may scroll down pretty fast, you can watch for * and + to easily track how often the program attempts to load the target page and parse HTML, respectively.

3.2. Uniqueness

To maintain previously seen hosts and IPs, you can use the following verification logic:

```
#include <unordered_set>
#include <string>
using namespace std;

//---------
DWORD IP = inet_addr ("128.194.135.72");
unordered_set<DWORD> seenIPs;
seenIPs.insert(IP);
...
//---------
unordered_set<string> seenHosts;

// populate with some initial elements
seenHosts.insert("www.google.com");
seenHosts.insert("www.tamu.edu");

string test = "www.cse.tamu.edu";
int prevSize = seenHosts.size();
seenHosts.insert (test);
```

```
if (seenHosts.size() > prevSize)
	// unique host
else
	// duplicate host
```

3.3. Page Buffers

Make sure to continue using a dynamic-buffering scheme for received pages in `Socket::Read`. Even though the maximum page size is now fixed, do not hardwire 2-MB buffers into your receiver. When you scale this program to 5000 threads in Part 3, such inefficient RAM usage may become problematic. Similarly, when you reuse the `Socket` class for the next connection, delete the old buffer if it is larger than 32 KB and start again with `INITIAL_BUF_SIZE`.

4. Description (Part 3)

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.

4.1. 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 stats should be printed:

```
Extracted 1000004 URLs @ 9666/s
Looked up 139300 DNS names @ 1346/s
Downloaded 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
```

4.2. 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) Obtain the average number of links per HTML page that came back with a 2xx code. Estimate the size of Google's webgraph (in terms of edges and bytes it occupies on disk) that contains 1T (trillion) crawled nodes and all of their out-links. Assume the graph is stored using adjacency lists, where each URL is represented by a 64-bit hash.

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/
```

4.3. 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.

4.4. 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 `Ini-`

tializeCriticalSection 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() hanging 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);
	}
}
```

4.5. 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://www.tamu.edu
        Parsing URL... host www.tamu.edu, port 80, request /
        Doing DNS... done in 4 ms, found 165.91.22.70
      * Connecting on page... done in 1 ms
        Loading... done in 57 ms with 27865 bytes
        Verifying header... status code 200
        Dechunking... body size was 27642, now 27595
      + Parsing page... done in 0 ms with 90 links

----------------------------------------
HTTP/1.1 200 OK
Date: Sun, 18 Jan 2015 23:12:17 GMT
Server: Apache/2.2.12 (Linux/SUSE)
Accept-Ranges: bytes
Keep-Alive: timeout=15, max=82
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: text/html
```

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.

4.6. Traces

The first example uses `URL-input-100.txt` and 10 threads (produced in 2015, which may differ from the results today):

```
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
Downloaded 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 1M URLs:

```
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
Hit 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
```

CSCE 463/612 Homework 1 Part 1

Name: _____

Function	Points	Break down	Item	Deduction
Input	1	1	No usage info if incorrect arguments	
Request	3	1	Incorrect GET syntax	
		1	No hostname in request	
		1	No user-agent in request	
Receive loop	4	1	No dynamic buffer resizing	
		2	Fails to receive/parse 96MB file	
		1	No select()	
Output	10	3	Incorrect host/port/request	
		1	Incorrect DNS info	
		1	No timing of connect()	
		1	No timing of recv()	
		1	Incorrect page size	
		1	Incorrect HTTP status	
		1	Incorrect number of links	
		1	Incorrect HTTP header shown	
Errors	6	1	Does not handle invalid port/scheme	
		1	Does not notify of DNS failure	
		1	Does not notify of connect failure	
		1	Does not notify of recv failure	
		1	Does not notify of non-HTTP reply	
		1	Parses non-2xx pages	
Other	1	1	Missing files for compilation	

Additional deductions are possible for memory leaks and crashing.

Total points: _____

CSCE 463/612 Homework 1 Part 2

Name: _____

Function	Points	Break down	Item	Deduction
Output	11	1	Fails to show input file size	
		1	Incorrect URLs being crawled	
		1	Incorrect DNS results	
		1	Fails to print host checks	
		1	Fails to print IP checks	
		1	No timing of robots connect()	
		1	No timing of robots recv()	
		1	Incorrect robots page size	
		1	Incorrect robots HTTP status	
		2	Incorrect page download results	
Logic	8	2	Fails to load multiple pages	
		2	Allows duplicate hosts	
		2	Allows duplicate IPs	
		2	Loads robots-prohibited pages	
Robot errors	5	1	Does not notify of connect failure	
		1	Does not notify of recv failure	
		1	Does not notify of non-HTTP reply	
		1	Fails to report slow download	
		1	Fails to report exceeding max	
Other	1	1	Missing files for compilation	

Additional deductions are possible for memory leaks and crashing.

Total points: _____

CSCE 463/612 Homework 1 Part 3

Name: _____

Function	Points	Break down	Item	Deduction
Running output	12	1	Printouts not every 2 seconds	
		1	Incorrect active threads	
		1	Incorrect queue size	
		1	Incorrect extracted URLs	
		1	Incorrect unique hosts	
		1	Incorrect DNS lookups	
		1	Incorrect unique IPs	
		1	Incorrect attempted robots	
		1	Incorrect crawled URLs	
		1	Incorrect parsed links	
		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: _____