

CPS 470/570: Computer Networks

Project #1 due 11:55 pm, 2/14/2018 (100 pts)

At most three students in a team. One submission per team

No late submission will be accepted

Receive 5 bonus points if turn in the complete work without errors at least one day before deadline

Receive an *F* for this course if any academic dishonesty occurs

1. Purpose

This homework builds an understanding of the Application Layer, Winsock programming, and multithreaded programming.

2. Description

Using Winsock and Visual Studio .NET, your goal is to create a web crawler. Your project will download multiple URLs from an input file using a single thread. To ensure politeness, you will need to **hit only unique IPs**. To avoid hanging up the code on slow downloads, you will also have to abort all pages that take longer than 10 seconds or occupy more than 2 MB (for robots, this limit is 16 KB).

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.

2.1. Single Threaded Crawling Code with URL-input-100.txt (40 pts)

The program must accept two arguments. The first argument indicates the number of threads to run and the second one the input file:

```
as1.exe 1 URL-input-100.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.

Make sure that only **unique hosts** make it to `gethostbyname` and that only **unique IP** make it to request:

Parse URL → Check host is unique → DNS lookup (to get IP address) → Check IP is unique → Request robots (to request a header only) → Check HTTP code → Request page (entire file) → Check HTTP code → Parse page

Note that robot existence should be verified using a **HEAD request**, which **ensures that you receive only a header rather than an entire file**. That is, in a HEAD request, your program requests the robot file, i.e., "/robots.txt" file. **Codes 4xx** indicate that the robot file does not exist and the website allows unrestricted crawling. Any other code should be interpreted as preventing further contact. For more information, here is the link: <http://www.robotstxt.org/orig.html>

Your printouts should begin with indication that you read the file and its size, followed by the following trace:

```
Opened URL-input-100.txt
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
```


Uniqueness-verification steps and the robot phase are highlighted in bold. You should have a function that connects to a server, downloads a given URL, and verifies the HTTP header. You can simply call this function 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 page-step, 4xx for robot-step**); 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.

Basic operation of Winsock is covered in class and sample code. Additional caveats are discussed next.

2.2. Required HTTP Fields

The general URL format is given by:

```

scheme://[user:pass@]host[:port][/path][?query][#fragment]

```

No need to download a page if scheme is https. No need to parse username/password in this assignment. You should extract host, port number, path, and query. For instance:

Given URL <http://cs.udayton.edu:467?addrbook.php>, parse URL... host is **cs.udayton.edu**, port is **467**, path is empty, query is **addrbook.php**.

Given URL <http://138.194.135.11?viewcart.php/>, parse URL... host **138.194.135.11**, port **80**, path is **/**, query is **viewcart.php**

HTTP request includes `[/path]` `[?query]`.

```
GET /some/page/index.php?status=15 HTTP/1.0
Host: udayton.edu
\r\n
```

As shown above, an HTTP request begins with the method line (**GET** or **HEAD** in this assignment, same syntax), HTTP/version, followed by (field: value) pairs (e.g., field is `Host`, value is `udayton.edu`), and ends with an empty line.

Requests may keep the connection open for some non-compliant servers, which makes it difficult to detect the end of transfer. We thus add “**Connection: close**” to explicitly request that the server close the connection:

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

You may invent your own string in the format of `crawlerName/x.y`, where `x.y` can evolve from 1.1 to 1.3.

When parsing an URL, find `#` to strip off the fragment. Find first `/`, `:`, `?` to extract path, port number, and query. See functions in string at <http://www.cplusplus.com/reference/string/string/?kw=string>

If the path is empty, you must use the root `/` in its place. **If query is empty**, do **not** add `?` in the request.

Next, insert your request into a character array and then call `send()`. The following is the outline.

```
string URL = "http://azlist.about.com/a.htm"; // a URL example
string host = "", path = ""; //You should parse URL to get host "azlist.about.com", path "a.htm"

string sendstring = "GET /" + path + " HTTP/1.0\nUser-agent: UDCScrawler/1.0\nHost: " + host
                  + "\nConnection: close" + "\n\n"; // pay attention to required white space

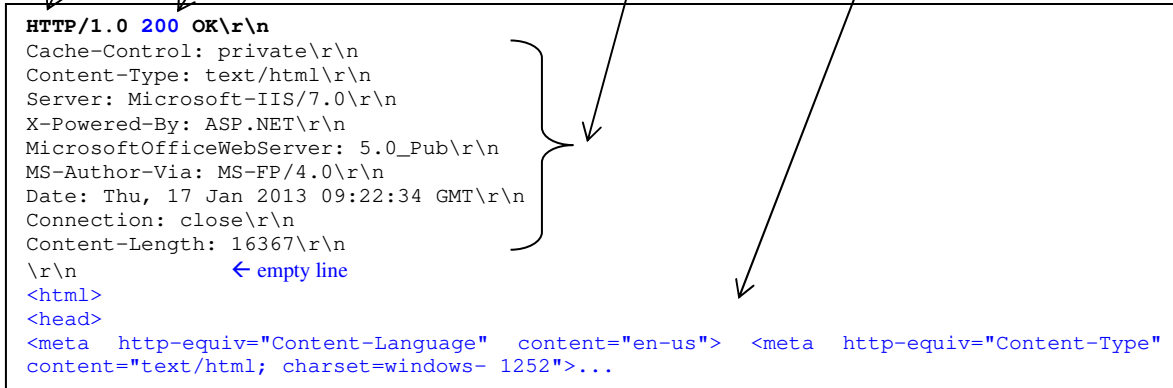
int size = sendstring.length(); // length of string, in terms of bytes

if (send(sock, sendstring.c_str(), size, 0) == SOCKET_ERROR)
{
    printf("send() error - %d\n", WSAGetLastError ());
    return false; // your function returns false
}
return true; // otherwise successfully send a GET request
```

In next subsection, we show how to use a loop to receive HTTP response.

2.3. Receive Loop

An HTTP response consists of status line, HTTP header, and then **object**. Status line begins with HTTP/, and status codes are 3-digit integers.



```
HTTP/1.0 200 OK\r\n
Cache-Control: private\r\n
Content-Type: text/html\r\n
Server: Microsoft-IIS/7.0\r\n
X-Powered-By: ASP.NET\r\n
MicrosoftOfficeWebServer: 5.0_Pub\r\n
MS-Author-Via: MS-FP/4.0\r\n
Date: Thu, 17 Jan 2013 09:22:34 GMT\r\n
Connection: close\r\n
Content-Length: 16367\r\n
\r\n
<html>
<head>
<meta http-equiv="Content-Language" content="en-us"> <meta http-equiv="Content-Type"
content="text/html; charset=windows-1252">...
```

The diagram illustrates the structure of an HTTP response. Arrows point from the text description to specific parts of the response: 'HTTP/' points to the status line, 'status codes' points to '200', 'HTTP header' points to the block of headers (Cache-Control, Content-Type, etc.), and 'object' points to the HTML body content.

The function below checks the socket to see if there is any data (via `select()`) before attempting a receive. Without doing this, you may experience deadlocks inside `recv()` call when the remote host neither provides any data nor closes the connection.

```
#define BUF_SIZE 1024    // array size
#define TIMEOUT 20000    // 20 seconds

class Winsock{
public:
    Winsock() { } // empty constructor
    ~Winsock() { } // destructor
    // --- many public methods ---
    bool Receive (string & recv_string);
    SOCKET sock;
private:
    char buf[BUF_SIZE]; // char array used to receive data from the server
}

bool Winsock::Receive( string & recv_string) //recv_string is modified after function is called
{
    FD_SET Reader; // for select() function call
    FD_ZERO(&Reader);
    FD_SET(sock, &Reader); // add your socket to the set Reader

    // set timeout, used for select()
    struct timeval timeout;
    timeout.tv_sec = TIMEOUT; // must include <time.h>
    timeout.tv_usec = 0;

    recv_string = ""; // initialized as an empty string, used to save all received data
    int bytes = 0; // count how many bytes received via each recv()
    do{
        if (select(0, &Reader, NULL, NULL, &timeout) > 0) // if have data
        {
            if ((bytes = recv(sock, buf, BUF_SIZE-1, 0)) == SOCKET_ERROR)
            {
                printf("failed with %d on recv\n", WSAGetLastError());
                return false;
            }
            else if (bytes > 0)
            {
                buf[bytes] = 0; // NULL terminate buffer
                recv_string += buf; // append to the string recv_string
            }
            // quit loop if it hits the maximum size, i.e., 2 MB for pages, 16 KB for robots
        }
    }
```

```

        else
        {
            // timed out on select()
            return false;
        }
    } while ( bytes > 0); // end of do-while
    return true;
} // end of Winsock::Receive

```

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

NOT required: An alternative to traditional `select()` is `WSAEventSelect()` or the IOCP framework ([http://msdn.microsoft.com/en-us/library/windows/desktop/aa365198\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/aa365198(v=vs.85).aspx)), which you can explore only if the rest of the homework appears too simple. The `WSAEventSelect` 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.

To parse a page, use function `recv_string.find(...)` to collect data, e.g., “**href**” for finding links this page has, and status code such as “200 OK” and “301”.

2.4. Other 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`. Search these functions at the website <http://www.cplusplus.com/> to see how to use them.

Oftentimes, it is convenient to declare a fixed-size buffer that is large enough to accept even the longest URL. If the input string violates either bound, you should reject it. An explicit check is required since the input file (.txt) contains random URLs obtained from the web.

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

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`.

2.5. 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 ("138.164.21.72");
unordered_set<DWORD> seenIPs;
seenIPs.insert (IP);
...
//-----

```

```
unordered_set<string> seenHosts;
// populate with some initial elements
seenHosts.insert("www.google.com");
seenHosts.insert("www.udayton.edu");
string test = "www.yahoo.com";
int prevSize = seenHosts.size();
seenHosts.insert(test);
if (seenHosts.size() > prevSize)
    // unique host
else
    // duplicate host
```

2.6. Page Buffers

Make sure to reuse the string `recv_string` in `Socket::Receive` for the next connection to a new host. Do not hardwire 2-MB buffers into your receiver. When you scale this program to 5000 threads in Part 2, inefficient RAM usage may become problematic.

3. Multi-threaded Crawling

We are finally ready to multi-thread this program and achieve significantly faster download rates. Due to the high volume of outbound connections in your project, your home ISP (e.g., Suddenlink, cam-pus 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 (e.g., udayton DNS server), which may be construed as **malicious flooding attacks**. In such cases, you can run your own DNS (simplifiedns.com) on your computer or you should send **a few DNS requests per minute**.

3.1. Code with input URL-input-1M.txt (30 points)

The command-line format remains the same as simple threaded crawling, but allows more threads:

```
asl.exe 3500 URL-input-1M.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. The general algorithm follows this outline:

```
int main(int argc char **argv)
{
    // parse command line args
    // initialize shared data structures & parameters sent to threads

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

    // wait for N crawling threads to finish
    // print stats data (below)
    // cleanup
}
```

Your crawler should process the following information:

```
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

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
```

3.2. Report (30 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 multithreads, 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) that contains 1T 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 Yahoo 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. (10 extra pts) How many of the crawled 2xx pages contain a hyperlink to our domain `udayton.edu`? How many of them originate from outside of udayton? Explain how you obtained this information.

3.3. 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 the Operating System course, 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 () // in main()
{
    // produce items, which are read from input file into the queue in AS#1
    for (i = 0; i < N; i++)
        Q.push(item[i]);
}
```



```

Consumer () // in crawling thread
{
    while (true)
    {
        critical {
        section {
            Lock mutex; // obtain mutex in order to modify shared Q
            if (Q.size() == 0) // finished crawling?
            {
                mutex.Unlock();
                break;
            }
            x = Q.front(); Q.pop();
            Unlock mutex;
        }

        parallel {
            // crawl x, collect stats data: valid host/IP? Status code? # of links found on page x? ...

            critical {
            section {
                Lock mutex; // to modify shared parameters
                // modify shared parameters: total # of unique hosts, total # of links, ...
                Unlock mutex;
            }
        }
    }
}

```

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` (below), but they are much slower.

```

// create a mutex for accessing critical sections (including printf); initial state = not locked
HANDLE mutex = CreateMutex (NULL, 0, NULL); // as in sample code

```

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`.

You will need to use mutex to synchronize on updating shared data structures, e.g.:

```

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

```

In addition, to avoid slowing down the user-interaction response time of your computer, you should set all your threads to the lowest priority:

```

SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_LOWEST);

```

3.4. Thread Issues

Starting too many threads may be difficult in 32-bit operating systems due to the large space needed in the kernel to handle thread control data. It is recommended that you use a system with at least 2 GB of RAM and a 64-bit operating system.

Here are several suggestions that will overcome problems with running out of thread memory in the kernel (which usually manifests itself in calls to `bad_alloc()` with out-of-memory errors in Debug mode). First, reduce the reserve stack size in the project using Visual Studio .NET 2013:

Project Properties->Linker->System->Stack Reserve Size = 65536

Second, use Windows Task Manager to see the number of threads actually running and make your code report any errors returned from `CreateThread` to the user. To see the thread count per process, use View->Select Columns in Task Manager.

3.5. DNS Lookup Issues

During crawling, you may generate a huge amount of traffic to your local DNS server (the default DNS server at my computer is `ns3.udayton.edu`) and potentially crash it. This may lead to suspicion that you are performing malicious activity and purposely trying to compromise network security. To find out your local DNS server, go to the command prompt and type `nslookup` without any arguments.

To avoid complications, you should reset the DNS server of your own computer using one of the following methods. You need *administrator* privileges to switch the DNS server.

Method 1 (for students who live on campus): Run DNS queries slowly, since we do not want to overwhelm UD DNS servers.

Method 2 (recommended for all students): you can install a free trial version of Simple DNS Plus (<http://www.simplesdns.com>). After you **install** DNS locally on your computer, you may set the local DNS server to **127.0.0.1**: Go into Network Connections->Local Area Connection (Properties)->Internet Protocol (TCP/IP) (Properties) and modify the field called "Preferred DNS server" by entering 127.0.0.1. Then `gethostbyname` will send all lookup requests to your local computer.

Method 3 (not recommended, but a working method): Use Google Public DNS at <https://developers.google.com/speed/public-dns/docs/using>. In this case, you do not need to install any DNS server at your computer, but run the lookups slowly on Google DNS.

To verify that DNS is working as intended, run `nslookup` at the command prompt.

3.6. Wireshark

Wireshark (<http://www.wireshark.org/>) is a software package that allows you to intercept all packets sent and received by your computer. Wireshark allows you to diagnose implementation problems encountered in this and other homework.

3.7. Debug vs. Release Mode

When using a large number of threads, always run your code in Release mode as it runs 50 times faster in STL functions and occupies 50% less memory. For scalar classes inserted into STL sets, you can roughly estimate 60 bytes per entry in Debug mode and 30 bytes in Release mode. If you insert other STL objects (such as strings, `unordered_set`) into a set, then count a minimum of 90 bytes per entry plus the length of the string in Debug mode and 55 bytes in Release mode.

Furthermore, to avoid swapping to disk and showing unacceptably low performance in your report, check that the total memory usage in Task Manager is well below your physical RAM size. You can notice that something is wrong when increasing the number of threads beyond some threshold (such as 2500) leads to significantly lower performance.