Assignment 4: RSS News Feed Aggregator

Virtually all major newspapers and TV news stations have bought into this whole Internet thing. What you may **not** know is that all major media corporations serve up RSS feeds summarizing the news stories that've aired or gone to press in the preceding 24 hours. RSS news feeds are XML documents with information about online news articles. If we can get the feeds, we can get the articles, and if we can get the articles, we can build an index of information similar to that held by `news.google.com`. That's precisely what you'll be doing for Assignment 4.

This assignment is the first of **two** threading assignments you'll get back to back. The first one will make sure you gracefully manage the transition from C to C++ while gaining some some experience with C++ containers like the `map` and the `vector`, C++11 `threads`, `mutexes`, and (our homegrown) `semaphores`. You'll receive a follow-up assignment a week from today—your Assignment 5—which will revisit some of the design decisions being imposed here while forcing you to be much more conservative about the number of actual threads you create over the lifetime of the program. Assignments 4 and 5 could have been one large assignment, but I decided to break it down and give them out as a pair of short, focused seven-day assignments instead.

Due: Friday, February 20th at 11:59 p.m.

Indexing a news article amounts to little more than breaking the story down into the individual words and noting how often each word appears. If a particular word appears a good number of times, then said word is probably a good indicator as to what the web page is all about. Once all of the world's online RSS news feeds have been indexed, you can ask it for a list of stories about a specific person, place, or thing.

If, for instance, you're curious what people are doing this Friday, you can query your friendly news index and it's sure to come back with something juicy:

```
myth22> ./news_aggregator --verbose --url large-feed.xml
// lots of logging output omitted
Enter a search term [or just hit <enter> to quit]: Halloween
```

```
    That term appears in 123 articles. Here are the top 15 of them:

        1.) ''Cops get spooky with surveillance video for Halloween''
[appears 17 times].

            ''http://www.philly.com/r?19=961&43=165761&44=280527492&32=3796&7=......html

        2.) ''Boston-Themed Halloween Costumes for the Lazy'' [appears 16
times].

            ''http://feeds.boston.com/c/35022/f/646891/s/3fe06d0a/sc/17/l/
0L0S.....1.htm''

        3.) ''Man shot outside Halloween party in Camden County'' [appears
12 times].

            ''http://www.philly.com/r?19=961&43=165761&44=280507862&32=3796&7=......html

        4.) ''Unseasonably warm today, but chills by Halloween'' [appears
10 times].

            ''http://feeds.chicagotribune.com/~r/chicagotribune/news/
nationwor.....1.htm''

        5.) ''Move Over Pumpkin Spice-We Have Reached Peak Candy Corn''
[appears 9 times].

            ''http://feeds.boston.com/c/35022/f/646891/s/3fe10b2c/sc/14/l/
0L0S.....1.htm''

        6.) ''A Place to Light the Fire Within'' [appears 7 times].

            ''http://feeds.boston.com/c/35022/f/646891/s/3fd82255/sc/10/l/
0L0S.....1.htm''

        7.) ''If Your Car Hits a Moose in Maine, Dibs on the Carcass''
[appears 7 times].

            ''http://feeds.boston.com/c/35022/f/646891/s/3fd8ded8/sc/36/l/
0L0S.....1.htm''

        8.) ''Boston Radio Legend Dale Dorman Dies'' [appears 7 times].

            ''http://feeds.boston.com/c/35022/f/646891/s/3fd8e91d/sc/8/l/
0L0Sb.....1.htm''

        9.) ''Bye-Bye Blue Laws: Now You Can Hit the Packie on Sunday''
[appears 7 times].

            ''http://feeds.boston.com/c/35022/f/646891/s/3fd99d63/sc/8/l/
0L0Sb.....1.htm''
```

```
       10.) ''6 Awesomely Bad Horror Movies Streaming on Netflix'' [appears
5 times].

            ''http://feeds.boston.com/c/35022/f/646891/s/3fe012b9/sc/36/l/
0Lbd.....1.htm''

       11.) ''U.S. not prepared to deal with widespread Ebola'' [appears 5
times].

            ''http://www.dispatch.com/content/stories/national_world/2014/
10/2.....html''

       12.) ''Forfeiture laws allow agents to steal money'' [appears 5
times].

            ''http://www.dispatch.com/content/stories/national_world/2014/
10/2.....html''

       13.) ''Drills gave NYC jump-start on Ebola'' [appears 5 times].

            ''http://www.dispatch.com/content/stories/national_world/2014/
10/2.....html''

       14.) ''Ebola nurse quarantined on arrival in U.S.'' [appears 5
times].

            ''http://www.dispatch.com/content/stories/national_world/2014/
10/2.....html''

       15.) ''Cuts in U.S., European defense spending stir fears'' [appears
5 times].

            ''http://www.dispatch.com/content/stories/national_world/2014/
10/2.....html''
```

If you’re contemplating a semester abroad, you might see what Beijing is up to these days:

```
    Enter a search term [or just hit <enter> to quit]: Beijing

    That term appears in 3 articles. Here they are:

        1.) ''Chen Ziming, jailed leader of China’s 1989 Ti.....at 62''
[appears 6 times].

            ''http://feeds.washingtonpost.com/c/34656/f/636708/s/3fda779a/
sc/3.....1.htm''

        2.) ''The SUV goes from All-American to global star'' [appears 4
times].
```

```
            ''http://seattletimes.nwsource.com/html/nationworld/
2024881964_apx.....n=rss''
        3.) '''Yellow Umbrella' app lets you play Hong Kong protester''
[appears 1 time].
            ''http://feeds.chicagotribune.com/~r/chicagotribune/news/
nationwor.....1.htm''
```

Some search terms are so focused (and our index is, in the grand scheme of indices, so small) that they produce just one result:

```
    Enter a search term [or just hit <enter> to quit]: Salvador
    That term appears in 1 article. Here it is:
        1.) ''An undocumented immigrant’s dream deferred'' [appears 1
time].
            ''http://feeds.washingtonpost.com/c/34656/f/636708/s/3fd4339f/
sc/1.....1.htm''
```

And very occasionally, something perfectly awesome doesn’t get mentioned at all:

```
    Enter a search term [or just hit <enter> to quit]: CS110
    Ah, we didn't find the term ''CS110''. Try again.
```

Files

Here’s a description of each of the files contributing to the overall code base we’re giving you:

`news_aggregator.cc`

`news_aggregator.cc` is the primary file that defines the main entry point and leverages the functionality provided by all other files to build the index and then allow the user query that index as we’ve illustrated above. We’ve already implemented the **main** function, and we’ve even implemented the `queryIndex` function for you. You’ll need to complete the implementation of the `processAllFeeds` function to build the news index so the provided `queryIndex` function has real data to chew on.

This is the only file you’ll likely modify, and we’ll describe what steps you need to take in a moment.

`news-aggregator-utils.h/cc`

`news-aggregator-utils.h` defines a small number of URL and string utility functions that contribute to the implementation of several functions in `news_aggregator.cc`. The implementation of `queryIndex` I already provide calls the `shouldTruncate` and `truncate` functions, and you'll ultimately want to call the `getURLServer` function as you implant thread count limits on a per-server basis.

You shouldn't need to change either of these files.

`article.h`

`article.h` defines a simple record—the `Article`—that pairs a news article URL with its title. `operator<` has been overloaded so that it can compare two `Article` records, which means that `Articles` can be used as the keys in STL `maps`.

You shouldn't need to change this file.

`html-document.h/cc`

The `html-document` files define and implement the `HTMLDocument` class, which models a traditional HTML page. It provides functionality to pull an HTML document from its server and surface its payload—specifically, the plain text content under its `<body>` tag—as sequence of token. The `parse` method manages the networking and processes the content, and the `getTokens` method provides access to the sequence of tokens making up the pages. All of the news articles referenced by the RSS feeds are standard web pages, so you'll rely on this `HTMLDocument` class to pull each of them and parse them into their constituent tokens.

You shouldn't need to change either of these files.

`rss-feed.h/cc`

The `rss-feed` files define and implement the `RSSFeed` class, which models a particular type of standardized XML file known as an RSS feed. The structure of an RSS feed document isn't important, since an `RSSFeed` object, when constructed around an URL of such a feed, knows how to pull and parse the XML document just as an `HTMLDocument` knows how to pull and parse an HTML document. The primary difference is that an `RSSFeed` produces a sequence of `Articles`, not a sequence of tokens. So, `RSSFeeds` produce `Articles` housing URLs which can be fed to `HTMLDocuments` to produce words.

You shouldn't need to change either of these files.

`rss-feed-list.h/cc`

The `rss-feed-list` files define and implement the `RSSFeedList` class, which models an other XML file whose format was invented for the purpose of this assignment. `RSSFeedList`'s story is similar to that for `RSSFeed`, except that it surfaces a feed-title-to-feed-url URL map.

You shouldn't need to change either of these files.

`rss-index.h/cc`

The `rss-index` files define and implement the `RSSIndex` class, which models the news article index we've been talking about. An `RSSIndex` index maintains information about all of the words in all of the various news articles that've been indexed. The code you add to `news_aggregator.cc` will directly interface with a single, global instance of this class, so you'll want to be familiar with it—in particular, you should inspect the `rss-index.h` file so you're familiar with the `add` method. Note that the `RSSIndex` implementation is not thread-safe.

You shouldn't need to change either of these files.

`*-exception.h`

These three header files each define and inline-implement a custom exception class that gets instantiated and thrown when some network drama prevents one of the three `parse` methods from doing its job. It's possible the URL was malformed, or it's possible your WiFi connection hiccuped and your network connection was dropped mid-parse. Most of you haven't dealt with exceptions much since CS106B/X, or maybe even CS106A. However, the starter code illustrates how `try/catch` clauses work: Just inspect the partial implementation of `processAllFeeds` we're starting you off with and you'll understand the idiom in a matter of 15 seconds.

You shouldn't need to change any of these header files.

`stream-tokenizer.h/cc`

The `stream-tokenizer` files provide the C++ equivalent of the Java `StreamTokenizer` class. The implementation is not pretty, but that's because it needs to handle UTF8-encodings of strings that aren't necessarily ASCII. Fortunately, you should be able to ignore this class, since it's really used to decompose the already implemented

`HTMLDocument` class. Feel free to peruse the implementation if you want, or ignore it. Just understand that it's there and contributing to the overall solution.

You shouldn't need to change either of these files.

Getting Code

Go ahead and clone the mercurial repository that we've set up for you by typing:

```
myth22> hg clone /usr/class/cs110/repos/assign4/$USER assign4
```

Compile often, test incrementally and almost as often as you compile, `hg commit` a bunch so you don't lose your work if someone reboots the `myth` machine you're working on, and run `/usr/class/cs110/tools/submit` when you're done.

Implementing v1: Sequential `news_aggregator`

Your final submission ultimately needs to provide a **multithreaded** version of `news_aggregator`, subject to a laundry list of constraints we'll eventually specify. You're free to implement the multithreaded version right from the start and forgo an intermediate, **sequential** version. My strong feeling, however, is that you should invest the time getting a sequential version working first (sans any threading) to make sure everything works predictably, that you've made proper and elegant use of all of the classes we've provided, and that your sequential application's output matches that produced by the sample (see note below). For what it's worth, this is precisely the approach I took as I developed my own solution, so don't go on thinking you're weak because you don't try to manage the full implementation in one pass. You know this already, but I'll say it again: Incremental development is the key to arriving at a robust, bug-free executable much more quickly.

Note: Honestly, you might see small differences between the sample application and your own, because a small fraction of an article's content—the advertising, actually—is dynamically generated for each download and therefore impacts how the index is built each time.

There are two things I would do while implementing the sequential version, however, and those two things are this:

- I would test with the `small-feeds.xml` file until you've convinced you've reached the sequential milestone. The huge drawback of the sequential version—in fact, a drawback I deliberately highlight so we later see why programming with threads is so incredibly powerful—is that it takes a long time to load each article one after another. The `small-feeds.xml` file should be good enough for the vast majority of your development until you're ready to take on the concurrency requirements, and at that point you can test the sequential version one last time by launching your `news_aggregator` executable against `large-feeds.xml`, walking away to read War and Peace from start to finish, and then coming back to see if everything's finally indexed properly. It just might be.

- I would leave the global `RSSIndex` instance in place, even if it feels a little dirty doing so. In practice, you would rarely go with a global object in a sequential program, but we all know we're paying it forward to the time when scores and scores of threads will be concurrently modifying and otherwise accessing the index, so it's fine to leave it in global space from the get-go.

Implementing v2: Multithreaded `news_aggregator`

Practically speaking, the sequential, singly-threaded implementation of `news_aggregator` is farcically, unacceptably bad. Users don't want to wait ten minutes while an application fires up, so you should ultimately introduce multithreading in order speed up things up. The sequential version of the aggregator is painfully slow, as each and every request for an article from some remote server takes on the order of seconds. If a single thread sequentially processes each and every article, then said articles are requested, pulled, parsed, and indexed one after another. There's no overlay of network stall times whatsoever, so the lag associated with network connectivity scales with the number of processed articles, and it's like watching paint dry in 100% humidity.

Each RSS news feed can be downloaded in its own child thread, and each article identified within each feed thread can be downloaded in its own grandchild thread as well. Each article, of course, still needs to be parsed and indexed. But much of the dead time spent just waiting for content to come over the wire can be scheduled to overlap. The news servers hosted up by the New York Times, the Philadelphia Inquirer, and the Chicago Tribue are industrial strength and can handle hundreds if not thousands of requests per minute (or at least I'm hoping so, lest Stanford IP addressed be blacklisted between now and the assignment deadline. We'll see, won't we?)

Naturally, multithreading and concurrency have their shortcomings. Each thread needs thread-safe access to the index, so you'll need to introduce some mutexes to prevent race conditions from compromising your data structures.

Rather than outline a large and very specific set of constraints, I'm granting you the responsibility of doing whatever it takes to make the application run more quickly without introducing any synchronization issues. The assignment is high on intellectual content—after all, this is the first time where an algorithmically sound application can break because of race conditions and deadlock—but honestly, there isn't much additional coding once you get the sequential version up and running. You'll spend a good amount of time figuring out where the sequential version should spawn off child threads, when those threads should spawn off grandchild threads, and how you're going to use concurrency directives to coordinate thread communication.

Here's the fairly short list of must-haves:

- Each news feed should be downloaded in its own child thread, though you should limit the number of news feed documents being downloaded at any one time to 10.
- Each news article should be downloaded in its own grandchild thread, though you should limit the number of open connections to any one server (e.g. `www.philly.com`) to 6. Even heavy duty servers have a hard time responding to a flash mob of requests. It's generally considered good manners to limit the number of active conversations with any one server to some small number, and I'm choosing that number to be 6. (Browsers are even more conservative and usually limit it to 2 or 3 by default). Note that implementing this will be tricky, but you should be able to maintain a global `map<string, unique_ptr<semaphore>>` (making it thread-safe to the extent you need to) and leverage the implementation strategies I use to implement the `oslock` and `osunlock` manipulators, the implementation of which can be viewed by looking in `/usr/class/cs110/local/src/threads/` at `ostreamlock.cc`.
- Limit the **total** number of article download threads executing at any one time to be 32. There's little sense in overwhelming the thread manager with a large thread count, so we'll impose the overall limit to be 32. In practice, this number would be fine tuned for the number of processers, the number of cores per processor, and performance analytics, but we're not going to allow this to evolve into some analytics assignment on how to optimize performance. You're optimizing more than enough by just introducing clever use of threading in the first place. (Assignment 5 will have you revisit this decision and implement something called a thread pool, which is all kinds of awesome).

- Ensure that you don't index the same article twice. Assume two articles are the same if their titles are the same and they they come from the same hostname (e.g. `www.philly.com`)
- Ensure that access to all shared data structures is thread-safe, meaning that there's zero chance of deadlock, and there are no race conditions whatsoever. Use `mutexes` to provide the binary locks needed to protect against race conditions within critical regions. Make sure that all locks are released no matter the outcome of a download (e.g. inside `catch` clauses as well).
- Ensure that the full index is built before advancing to the query loop.
- Ensure that all memory is freed when the full application exits. If you elect to dynamically allocate memory (as you almost certainly will need to with your per-server semaphore `map`), then make sure that's all properly donated back to the heap before the `main` function returns.
- You **must** add logging code to emulate the logging provided by the sample executable, although you needn't format it precisely the same way. We'll be testing your solutions using the `--quiet` flag, but you should add logging just so you can see how much faster everything runs once you introduce threading.
- You may not change the implementation of `queryIndex`. It outputs the results in the exact format we'll expect to be used by the auto-grading process.

If you take the care to introduce threads as I've outlined above, then you'll speed up the configuration of your `news_aggregator` dramatically! Actually, dramatically isn't dramatic enough of a word to describe how dramatic the speedup will be. (Oh, you'll also get genuine experience with networking and the various concurrency patterns that come up in networked applications. That's important, too!)

I hope you're as excited about this assignment as I am! Leaping into the world of threads and race conditions can be scary, but the end result is so incredibly uplifting that it's more than worth the investment.