Assignment 5: RSS News Feed Aggregation

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 5.

This quarter's Assignment 5 intended to be the least intense of the quarter, and will gracefully manage your transition into multithreading while gaining some experience with C++ **`threads`**, **`mutexes`**, and **`semaphores`**. Assignment 6, which goes out a week from today, will have you revisit some of the Assignment 5's design decisions and force you to be much more conservative about the number of actual threads you create over the program's lifetime. (Oh! See this extra handout for tips on some C++ features you'll want to know when coding this one up.)

Due: Wednesday, November 8th 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 seeking updates about yesterday's vehicular terrorism in New York City, you might directly poll your index for all articles matching Manhattan, knowing the top articles are likely to be breaking news updates on the attack.

```
myth4> ./slink/aggregate_soln --url medium-feed.xml

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

That term appears in 122 articles.  Here are the top 15 of them:

   1.) "Eight dead in truck attack on Manhattan bike path; suspect arrested"
[appears 28 times].

       "http://feeds.reuters.com/~r/Reuters/domesticNews/~3/Bcf3RK5GM-s/
e.....D02QU"

   2.) "New York attack: Mayor Bill de Blasio says 'we will be undeterred'"
[appears 7 times].

       "http://www.bbc.co.uk/news/world-us-canada-41826402"
```

3.) "New York man pleads guilty to 'Hamilton' show ticket scheme" [appears 5 times].

"http://feeds.reuters.com/~r/Reuters/domesticNews/~3/eWpj422SyY0/n.....D032G"

4.) "Puerto Rico calls on U.S. utilities to help restore power" [appears 3 times].

"http://feeds.reuters.com/~r/Reuters/domesticNews/~3/2OlAXWVdswA/p.....D02FJ"

5.) "California insurance agency: wildfires losses at $3.3 billion, rising" [appears 3 times].

"http://feeds.reuters.com/~r/Reuters/domesticNews/~3/88aLFxcE9fI/c.....D02UT"

6.) "Former President Obama seeks to build U.S. civic engagement at summit" [appears 3 times].

"http://feeds.reuters.com/~r/Reuters/domesticNews/~3/Uh9vJHHbcGU/f.....D035K"

7.) "Man arrested in Utah shooting of student from China: police" [appears 3 times].

"http://feeds.reuters.com/~r/Reuters/domesticNews/~3/shqOg8ML9Cg/m.....D01UK"

8.) "New York attacker was 'screaming in the street'" [appears 3 times].

"http://www.bbc.co.uk/news/world-us-canada-41826038"

9.) "Footage shows New York suspect tackled by police" [appears 3 times].

"http://www.bbc.co.uk/news/world-us-canada-41826174"

10.) "White House: Trump campaign gave Papadopoulos emails to special counsel" [appears 2 times].

"http://feeds.reuters.com/~r/Reuters/PoliticsNews/~3/AByJsU6neRo/w.....D02PG"

11.) "Release of Republican U.S. tax bill delayed until Thursday: Axios" [appears 2 times].

"http://feeds.reuters.com/~r/Reuters/PoliticsNews/~3/JqtJqATjQ20/r.....D035O"

12.) "U.S. nuclear arsenal to cost $1.2 trillion over next 30 years: CBO" [appears 2 times].

"http://feeds.reuters.com/~r/Reuters/PoliticsNews/~3/SRf6gzIZPn8/u.....D030E"

13.) "Democrats want a law to stop Trump from bombing North Korea" [appears 2 times].

```
       "http://feeds.reuters.com/~r/Reuters/PoliticsNews/~3/beGqvCeFQDE/
d.....D02IG"

  14.) "Trump chief of staff's Civil War comment sparks criticism" [appears 2
times].

       "http://feeds.reuters.com/~r/Reuters/PoliticsNews/~3/d-XAjUNsdz8/
t.....D01WF"

  15.) "U.S. Senate Republicans want to speed Trump nominee approvals" [appears
2 times].

       "http://feeds.reuters.com/~r/Reuters/PoliticsNews/~3/i55mN-8maVU/
u.....D02N5"
```

Some search terms may have been relevant a few days ago, but have already been catalogued as yesterday's news and don't surface all that much:

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

That term appears in 2 articles.  Here they are:

   1.) "Former Trump campaign adviser denies encouraging aide on Russia
dealings" [appears 4 times].

       "http://feeds.reuters.com/~r/Reuters/PoliticsNews/~3/ynxCzUXWALs/
f.....D02MT"

   2.) "George Papadopoulos: Trump trashes 'low level' indicted aide" [appears
4 times].

       "http://www.bbc.co.uk/news/world-us-canada-41820677"
```

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

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

Ah, we didn't find the term "Stanford".  Try again.
```

Milestone 1: Implementing Sequential `aggregate`

Your `NewsAggregator` class ultimately needs to provide a **multithreaded** version of `processAllFeeds`, 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 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 of `NewsAggregator` 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: you might see small differences between the sample application and your own, because a small fraction of an article's content—the advertising, most often—is dynamically generated for each download and therefore impacts how the index is built each time. The `sanitycheck` tool, however, has been seeded with static RSS feeds and HTML news articles, so you should rely on it for a reliable test framework.

There are a details that apply even to the sequential version, so I'll address them right here.

- Ensure that you never ever download the same exact URL twice. The feed lists and RSS feeds may surface the same URL more than once (e.g. the New York Times's world-news and religion feeds may each include the same URL to an article about the Vatican), so you need to ensure that you detect the duplicates. You should download an article the first time you see its URL, but ignore all of the others. If you try to pull a document and the networking fails for whatever reason (i.e. `RSSFeed::parse` or `HTMLDocument::parse` throws an exception), you shouldn't try to download it a second time, and you should still ignore any other repeated requests to download the same URL during program execution. Long story short: every URL gets one chance to contribute and that's it.
- When two or more HTML articles share the same title and are hosted by the same server, don't assume they're true duplicates. It's safe to assume they're all really the same article, but download all of them anyway, compute the running intersection of their token sets, and file the token intersection under the lexicographically smallest URL. So if, for instance, `http://www.jerrynews.com/a.html`, `http://www.jerrynews.com/b.html`, and `http://www.jerrynews.com/c.html` all lead to articles titled `"Rock Me Like a Jerry Cain"`, and independent downloads produce token vectors of `["a", "a", "b", "c", "a", "a"]`,`["a", "a", "b", "b", "a"]`, and `["a", "a", "b", "b", "c", "a", "d", "a"]`, respectively, ensure that the `RSSIndex` gets `"Rock Me Like a Jerry Cain"` under `http://www.jerrynews.com/a.html` with `["a", "a", "a", "b"]` as tokens. This technique overcomes the fact that many news articles have dynamically generated advertising content that can be intersected away via multiple downloads. Implementing this will require you investigate the `sort` and `set_intersection` functions, iterators, and/or `back_inserter`s (at least these are the C++'isms I used to implement this part). Your repo includes a standalone program in `test-union-and-intersection.cc` that serves no other purpose that to illustrate how these STL built-ins work.

And here's some advice and words of caution.

- I would test with the `small-feed.xml` file until you're 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-feed.xml` file should be good enough for the vast majority of your development needs until you're ready to take on the concurrency requirements. At that point you can test the sequential version one last time by launching your `aggregate` executable against `medium-feed.xml`, walking away

to read *War and Peace* from start to finish, and then coming back to see if everything's been properly indexed . It just might be.

- Other feeds to test with:
  - `static-alphabet-feed.xml`, which is a contrived feed that'll help confirm you properly coded up the requirement that the same URL is downloaded at most once, and that all of your token intersection logic is spot on.
  - `single-server-source-feed.xml`, where all HTML articles are drawn from the same exact server. This is really good for testing your per-server limits.
- Don't be alarmed if you see spurious parsing errors while testing (e.g. messages like "Operation in progress" come up quite a bit). If you're testing using `small-feed.xml`, then you're testing with live data, so you need to be super tolerant of the fact that some of the XML and HTML documents aren't perfectly structured. (The static feeds used by the `sanitycheck` tool have been curated to rely on articles that parse without drama much more often.)

Milestone 2: Implementing Multithreaded `aggregate`

Practically speaking, the sequential, singly-threaded implementation of `aggregate` is unacceptably slow. Users don't want to wait 45 minutes for an application to fire up, so you should ultimately introduce multithreading in order to speed things up. The sequential version of `aggregate` is slow because each and every request for an article from some remote server takes on the order of seconds. If a single thread of execution 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.

Your `NewsAggregator` should be updated so that each RSS news feed is downloaded in its own child thread, and each article identified within each feed thread can be downloaded in its own grandchild thread. 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 BBC, Reuters, and The Economist are industrial strength and can handle hundreds if not thousands of requests per minute (or at least I'm hoping so, lest Stanford IP addresses be blacklisted between now and the assignment deadline).

Naturally, multithreading and concurrency have their shortcomings. Each thread needs exclusive access to the index while doing surgery, so you'll need to introduce some `mutex`es 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 such threads that can exist at any one time to 5.
- Each news article should be downloaded in its own grandchild thread, though you should limit the number of threads maintaining connections to any one server (e.g. `www.economist.com`) to 8. Even heavy duty servers have a hard time responding to a flash mob of requests, so it's considered good networking etiquette to limit the number of active conversations with any one server to some small number. (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 introduce a

  `std::map<std::string, std::unique_ptr<semaphore>>` to the private section of `NewsAggregator` (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`. (If the

  `std::map<std::string, std::unique_ptr<semaphore>>` proves to be too much work for you, then you can just go with `std::map<std::string, semaphore *>` instead.)
- Limit the **total** number of article download threads executing at any one time to be 18. (And that's 18 total, not 18 per feed thread.) There's little sense in overwhelming the thread manager with a large thread count, so we'll impose the overall limit to be 18. In practice, this number would be fine tuned for the number of processors, the number of cores per processor, and performance analytics, but no need to worry about that. Just assume that 18 is the right number.
- Ensure that you never ever download—or even **attempt** to download—the same exact URL twice. This was stated above during the specification of the sequential version, but it's particularly important here, since two threads might initiate a download of the same URL at the same time unless you implant measures to forbid that. As it turns out, the XML and HTML parsing libraries don't deal well when asked to pull the same URL in two simultaneously executing threads, so make sure you don't.
- Manage articles with the same title and server as you did for the sequential part, being sensitive the to the possibility that two such articles may be downloaded simultaneously in parallel threads.
- 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 `mutex`es 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 you allocate 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 can ignore any memory errors that stem from the XML library we rely on for parsing. It gets the job done, but it clearly wasn't written by a CS110 graduate/ :))

- You **can** add logging code to emulate the logging using the provided `NewsAggregatorLog` class, but you aren't required to. We'll be testing your solutions using the `--quiet` flag, but you can take delight if you add logging just so you can see how much faster everything runs once you introduce threading
- You may not change the implementation of the `NewsAggregator::queryIndex` method. It outputs the results in the exact format we'll expect to be used by the autograding process, so changing it would be a clear path to a 0, which would itself make the news.
- In contrast to Assignment 4, your error messages needn't match ours.

If you take the care to introduce threads as I've outlined above, then you'll dramatically speed up the configuration of your `aggregate` executable! You'll also get genuine experience with networking and the various concurrency patterns that come up in networked applications.

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/assign5/$USER assign5
```

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. There's also a sample executable called `aggregate_soln` in `/usr/class/cs110/samples/assign5` that you can run to see what it does, and we provide a link to it via the `slink` entry you've seen in all of your starter repos.

Files

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

`aggregate.cc`

`aggregate.cc` is a super small file that defines the `main` function and nothing else. The file is so small that I present it here:

```
int main(int argc, char *argv[]) {

  unique_ptr<NewsAggregator> aggregator(NewsAggregator::createNewsAggregator(argc,
argv));

  aggregator->buildIndex();
```

```
  aggregator->queryIndex();
  return 0;
}
```

You shouldn't need to change this file.

`news-aggregator.h/cc`

The `news-aggregator` files define the class used to build and query an index. You'll update the `news-aggregator` interface and implementation files to implement the version of the assignment that allocates new threads without bound. Assignment 6 will ask that you define something class a ThreadPool, which recycles existing threads to download many articles instead of just one.

You'll definitely be changing these two files over the course the assignment.

`test-union-and-intersection.cc`

`test-union-and-intersection.cc` is there so you can see sample code on how to use the C++ algorithms most useful for managing token set intersections. As written, this standalone program doesn't directly contribute to your `aggregate` executable, but it's there as a reference so you have an idea how you should be using `vectors`, `sort`, and `set_intersection`.

You're free to tinker with this file if you'd like to, but you're not required to make any changes whatsoever.

`utils.h/cc`

`utils.h` and `.cc` define and implement a small number of URL and string utility functions that contribute to the implementation of several functions in your `NewsAggregator` class. The provided implementation of `queryIndex` 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, although you're welcome to if it helps you arrive at a working program more quickly.

`log.h/cc`

`log.h` defines a small `NewsAggregatorLog` class that helps manage all of the progress and error messages that come up during program execution, and `log.cc` implements them. You can read through the interface and implementation files to figure out what method is likely called where within working implementations of `buildIndex` and `processAllFeeds`. In fact, the logging might help you work through the workflow of program execution as you implement, test, and debug. You, however, are **not required** to use the logging facilities provided by the `NewsAggregatorLog` class if you don't want to.

`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 `Article`s can be used as the keys in STL `map`s.

You shouldn't need to change either of these files, although you're welcome to if it helps you arrive at a working program more quickly.

`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 a sequence of tokens. 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, although you're welcome to if it helps you arrive at a working program more quickly.

`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 a 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 `Article`s, not a sequence of tokens. So, `RSSFeed`s produce `Article`s housing URLs which can be fed to `HTMLDocument`s to produce words.

You shouldn't need to change either of these files, although you're welcome to if it helps you arrive at a working program more quickly.

`rss-feed-list.h/cc`

The `rss-feed-list` files define and implement the `RSSFeedList` class, which models another XML file type 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, although you're welcome to if it helps you arrive at a working program more quickly.

`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. I guarantee the code you add to your `NewsAggregator` class interacts with an `RSSIndex`, 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.

My own solution actually changed these two files to codify the rules on how to handle articles with the same title but different URLs on the same server. You should consider doing the same thing.

`*-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. Recall that functions and methods capable of throwing exception can be optionally placed under the jurisdiction of a `try/catch` block, as with this:

```
HTMLDocument document(article.url);
try {
  document.parse();
} catch (const HTMLDocumentException& hde) {
  log.noteSingleArticleDownloadFailure(article);
  return;
}
// got this far? then article was downloaded just fine
```

You shouldn't need to change any of these header files, though you're welcome to if you'd like.

`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.

As always, compile more often than you blink, test incrementally, and `hg commit` with every bug-free advance toward your final solution. Of course, be sure to run `/usr/class/cs110/tools/submit` when you're done, and invoke that `sanitycheck` a whole lot.

Grading

Your assignments will be exercised using the tests we've exposed, plus quite a few of others. I reserve the right to add tests and change point values if during grading I find some features aren't

being exercised, but I'm fairly certain the breakdown presented below will be a very good approximation regardless.

News Aggregator Tests (82 points)

- Clean Build: 2 points
- Ensure that your aggregator handles a small feed list: 30 points
- Ensure that your aggregator handles a medium feed list: 20 points
- Ensure that your aggregator handles a large feed list: 20 points
- Ensure that you properly handle duplicate URLs and articles with the same title hosted by the same server: 10 points