

Please note that some of the resources used in this assignment require a Stanford Network Account and therefore may not be accessible.

CS107
Spring 2008

Handout 27
May 9, 2008

Assignment 6: RSS News Search Revisited

Assignment 4 was all about using a collection of custom data types and generic containers to build a fast, super-lean, industrial-strength application. For the most part, the application does exactly what we want: it's set up to download every RSS feed on the planet, it ignores words that are sure to be in every article, it skips over inlined JavaScript, CSS rules, and HTML comments, and it associates every meaningful word with every article it appears in. What's the one feature that in practice would be deemed unacceptable? All articles are downloaded sequentially, and configuration takes on the order of several minutes instead of a few seconds.

Your high task this week has you leverage off of CS107-specific material in order to make the RSS news feed aggregator load faster and more effectively. This time, you're going to take my Assignment 4 solution (where all articles are downloaded one after another in sequence) and infuse a little thread and semaphore action so that hundreds of news articles are downloaded simultaneously.

Due: Monday, May 19th at 11:59 p.m.

Multithreaded RSS News Aggregator

In practice, the sequential, single-threaded implementation of Assignment 4's RSS News Feed Aggregator would be considered unacceptably slow. Users don't want to wait five to ten minutes while an application fires up. You're to take my own solution to Assignment 4 and introduce multithreading in order speed up the process that builds the set of indices. The sequential version of the aggregator is painfully slow, because 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 overlap of network stall times whatsoever, so the lag associated with network connectivity starts to build up. It's like watching paint dry.

Each news article can be pulled and indexed in its own thread. Each article still needs to be parsed and indexed, and that all happens on the client end. But all of the dead time spent waiting for documents to be served up by remote servers can be scheduled to overlap. The news servers hosted up by the BBC, the Philadelphia Inquirer, and the Chronicle are heavy duty and can handle scores if not hundreds of requests per minute (or at least I'm hoping so, lest Stanford be banned from downloading articles. We'll see, won't we?)

Concurrency has its shortcomings, of course. Each needs MT-safe access to a master stop word list, a master list of previously downloaded articles, and a master set of indices. You'll need to introduce some semaphores to prevent race conditions from rearing their ugly heads. I'd like you to take the initiative to transform my sequential version into your very own multithreaded version.

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 coding at all. You'll spend a good amount of time figuring out where the sequential starter application should fork off threads, and how you're going to use Semaphores to coordinate thread communication. But I think you'll ultimately agree that the assignment doesn't require much in terms of coding.

At the very least, you must:

- Ensure that each of the news articles are concurrently downloaded in its own thread. (You needn't spawn threads to parse the RSS feeds, just the HTML articles listed within them.) These child threads will parse the news article and add each word/article pair to the share **hashset** of indices. You can add the word/article pairs to the set if indices directly, or you can populate a local **hashset** with all pairs, and then map over it to dump everything into the master **hashset** just before the thread exits.
- Ensure that access to the shared data structures is MT-safe, meaning that there's zero chance of deadlock, and there are no race conditions whatsoever. Use **Semaphores** to provide the binary locks needed to protect against race conditions. Make sure that all locks are released no matter the outcome of a download.
- Ensure that the full index set is built before advancing to the query loop.
- Ensure that all memory is freed when the full application exits.
- Limit the number of simultaneously open HTTP connections. UNIX limits the number of open file descriptors per process to something like 128 or 256, so you want to be sure the number of open file descriptors (those embedded within your **urlconnections**) is always less than that. Use a semaphore as a generalized counter, and initially set it equal to 24. Before each call to **URLConnectionNew**, each thread pulling HTML news articles waits on that generalized counter to make sure that it's only one of 24 privileged threads trying to open a network connection. Once the thread has pulled and indexed all content, it should call **URLConnectionDispose** (the sequential solution does already: this releases the embedded **FILE ***) and then signal the generalized counter so that other blocked HTML threads can proceed.
- 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. Include directives to limits the number of connections to any single server to some small number (like 6 or 8). The implementation of this can be quite tricky, but can be simplified by introducing another `hashset` to the system.

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

If you finish this up rather quickly, you can always email Jerry for some nifty extension ideas.

Starter Files

As usual, there's a starter code directory, so you won't be surprised to hear there's a directory called `/usr/class/cs107/assignments/assn-6-rss-news-search`.

`cp -r` that directory over to your local file space. (There's a parallel directory which contain library code, but you needn't [and in fact probably shouldn't] copy that over.)

Electronically submit your solution using the `/usr/class/cs107/bin/submit` script.