Assignment 5: RSS News Aggregator Revisited

You've spent the last week building a snazzy, multithreaded RSS News Feed Aggregator—using as many threads as you wanted to—to concurrently download news articles from around the globe and build a **`news.google.com`**-like search index. This time, you're going to revisit the **`news-aggregator`** executable, but instead of employing an unbounded number of threads, you're going to rely on a limited number of them—on the order of 20—to do everything. This new approach relies on the notion of a **thread pool**, which is what you'll spend the vast majority of your time implementing this week.

Due: Wednesday, November 12th at 11:59 p.m.

The Problem With Assignment 4

Assignment 4 uses **`semaphores`** to limit the number of threads that can be in the middle of a **`parse`** call at any one time, but we didn't technically require you limit the number of threads that could exist at any one time. If some unfortunate scheduling allows, say, hundreds of threads to be created over the lifetime of the download cycle and exist simultaneously, the indexing phase of **`news-aggregator`** would exceed the maximum number of threads allowed to exist within a single process. (Don't believe a limit exists? Just write a short program that creates 500 threads in a **`for`** loop. Then try running it and see what happens).

Even if we *could* guarantee that no more than, say, 32 threads ever existed at any one time (which is possible if you go with that second version of **`semaphore::signal`** introduced in the **`myth-buster-concurrent`** example), there's no sense creating a thread to do some work only to let it die if we're going to need *another* thread to do pretty much the same thing later on. Building up and tearing down a thread is a relatively expensive process, so we should prefer a smaller number of threads that live very long lives to a larger number of threads that live very short ones.

An industrial-strength aggregator needs to mitigate competing requirements (employing a bounded number of threads while getting everything off of the main thread as quickly as possible), and nothing we did in Assignment 4 does that.

Here's a new idea: implement a **`ThreadPool`** class, which exports the following **`public`** interface:

```
class ThreadPool {

public:

    ThreadPool(size_t numThreads);

    void schedule(const std::function<void(void)>& thunk);

    void wait();

    ~ThreadPool();
};
```

Here is a simple program that uses a `ThreadPool` to execute a collection of 10 functions calls using 4 threads.

```
static const size_t kNumThreads = 4;
static const size_t kNumFunctions = 10;
int main(int argc, char *argv[]) {
    ThreadPool pool(kNumThreads);
    for (size_t id = 0; id < kNumFunctions; id++) {
        pool.schedule([id] {
            cout << oslock << "Thread (ID: " << id << ") has started."
                 << endl << osunlock;
            size_t sleepTime = (id % 3) * 100;
            sleep_for(sleepTime);
            cout << oslock << "Thread (ID: " << id << ") has finished."
                 << endl << osunlock;
        });
    }
    pool.wait(); // block until all scheduled functions have executed
    cout << "All done!" << endl;
    return 0;
}
```

The output of the above program might look like this:

```
myth9> ./thread-pool-test
Thread (ID: 3) has started.
Thread (ID: 2) has started.
Thread (ID: 1) has started.
Thread (ID: 0) has started.
Thread (ID: 0) has finished.
Thread (ID: 4) has started.
Thread (ID: 1) has finished.
Thread (ID: 5) has started.
Thread (ID: 2) has finished.
Thread (ID: 6) has started.
Thread (ID: 3) has finished.
Thread (ID: 7) has started.
Thread (ID: 7) has finished.
Thread (ID: 8) has started.
Thread (ID: 4) has finished.
Thread (ID: 8) has finished.
Thread (ID: 9) has started.
Thread (ID: 5) has finished.
Thread (ID: 9) has finished.
Thread (ID: 6) has finished.
All done!
myth9>
```

In a nutshell, the program's `ThreadPool` creates a small number of worker threads (in this example, four of them) and relies on those four workers to collectively execute all of the scheduled functions (in this example, 10 of them). Yes, yes, we could have spawned ten separate threads and not used a thread pool at all, but that's the unscalable approach your Assignment 4 went with, and we're trying to improve on that by using a **fixed** number of threads to maximize parallelism without overwhelming the thread manager.

Task 1: Implementing the `ThreadPool class`, v1

How does one implement this thread pool thing? Well, your `ThreadPool` constructor—at least initially—should do the following:

- launch a single *dispatcher* thread like this (assuming `dt` is a `private thread` data member):

  ```
  dt = thread([this]() {
      dispatcher();
  });
  ```

- launch a specific number of *worker* threads like this (assuming `wts` is a `private vector<thread>` data member):

  ```
  for (size_t workerID = 0; workerID < numThreads; workerID++) {
      wts[workerID] = thread([this](size_t workerID) {
          worker(workerID);
      }, workerID);
  ```

```
    }
```

The implementation of `schedule` should append the provided function pointer (expressed as a `function<void(void)>` , which is the C++11 way to type a function pointer that can be invoked without any arguments) to the end of a queue of such function pointers. Each time a zero-argument function is scheduled, the dispatcher thread should be notified. Once the implementation of `schedule` has notified the dispatcher that the queue of outstanding functions to be executed has been extended, it should return right away so even *more* functions can be scheduled right away.

The implementation of the private `dispatcher` method should loop interminably, blocking within each iteration until it has confirmation the queue of outstanding functions is nonempty. It should then wait for a worker thread to become available, select it, mark it as unavailable, dequeue the least recently scheduled function, put a copy of that function in a place where the selected worker (and **only** that worker) can find it, and then signal the worker thread to execute it.

The implementation of the private `worker` method should also loop interminably, blocking within each iteration until the dispatcher thread signals it to execute a previously scheduled function. Once signaled, the worker should go ahead and call the function, wait for it to execute, and then mark itself as available so that it can be discovered and selected again (and again, and again) by the dispatcher.

The implementation of `wait` should block until all previously-scheduled-but-yet-to-be-executed functions have been executed. The `ThreadPool` destructor should wait until all previously-scheduled-but-yet-to-be-executed **thunks** have executed to completion, somehow inform the dispatcher and worker threads to exit (and wait for them to exit), and then otherwise dispose of all `ThreadPool` resources. (Functions that take no arguments at all are called **thunks**. The `function<void(void)>` type is a more general type than `void (*)()`, and can be assigned to anything invokable—a function pointer, or an anonymous function—that doesn't require any arguments.)

Your `ThreadPool` implementation shouldn't orphan any memory whatsoever.

Task 2: Reimplementing `news-aggregator`

Once you have a working `ThreadPool`, you should reimplement `news-aggregator`, using two global `ThreadPools`. Your `assign5` folder includes `news-aggregator.cc` in more or less the same state it was initially presented to you for Assignment 4. You'll need to add code that downloads all of the news articles to compile an index so that `queryIndex` can return meaningful results.

Why two `ThreadPools` instead of just one? I want one `ThreadPool` (of size 6) to manage a collection of worker threads that download RSS XML documents, and I want a second `ThreadPool` (of size 12) to manage a second group of workers dedicated to news article downloads. In fact, your set of global variables should be exactly this:

```
static const size_t kFeedsPoolSize = 6;
```

```
    static const size_t kArticlesPoolSize = 12;

    static bool verbose = false;

    static ThreadPool feedsPool(kFeedsPoolSize);

    static ThreadPool articlesPool(kArticlesPoolSize);

    static RSSIndex index;

    static mutex indexLock;
```

To simplify the implementation, you should not worry about limiting the number of simultaneous connections to any given server, and you shouldn't worry about indexing the same article more than once. The two `ThreadPool`s you're using are pretty small, so it's pretty much impossible for any single news server to feel abused by the new `news-aggregator` implementation. And while I could require you to guard against duplicate articles (those with identical server/title pairs) like I did for Assignment 4, I'm not going to require it this time, because the code that does that doesn't rely on thread pooling, so I don't see the need to make you write it again. My primary interest is showing you that a constant number of threads can more or less accomplish what an unbounded number of threads accomplished for `news-aggregator` 1.0.

Note that you should still submit code for Assignment 4 that meets the Assignment 4 specification. That fact that I'm now requiring a new approach this time around shouldn't impact how you finish up the assignment due tonight.

Task 3: Optimizing the `ThreadPool` class, v2

Once you press through your `news-aggregator` reimplementation, I want you to go back and update your `ThreadPool` implementation to be a little less aggressive about spawning worker threads at construction time.

Consider the scenario where you create a `ThreadPool` of size 32, but the surrounding executable only schedules two or three functions every few seconds. If the functions are relatively short and execute quickly, the vast majority of the worker threads spawned at construction time will be twiddling their thumbs with nothing to do. There's no sense creating worker threads and burdening the thread manager with them until they're really needed.

To remedy this, you should update your `ThreadPool` implementation to *lazily* spawn a worker thread only when you need a worker and all existing workers are busy doing something else. (Note that a worker thread isn't truly spawned until a thread with an installed thread routine is moved into the `wts vector`.)

You should still ensure the number of worker threads never exceeds the thread pool size, but you should only spawn worker threads on an as-needed basis. (Once you spawn a worker thread, it can exist forever, even if it never gets used again).

Files

The files we're giving you comprise a proper superset of the files we gave you for Assignment 4. The new files are:

`thread-pool.h/cc`

`thread-pool.h` presents the interface for the `ThreadPool` class, which you are responsible for implementing. Naturally, you'll want to extend the `private` section of the class definition to give it more state and the set of helper methods needed to build it, and you'll need to flesh out the implementations of the `public` methods in `thread-pool.cc`.

`thread-pool-test.cc`

`thread-pool-test.cc` contains the trivial program I posted above, and can be used (and extended) to fully exercise your `ThreadPool` class.

Getting The Code

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

```
    myth9> hg clone /usr/class/cs110/repos/assign5/$USER assign5
```

The code base you're cloning is more or less identical to that you cloned for Assignment 4, save for the fact that there are three more files: `thread-pool.h`, `thread-pool.cc`, and `thread-pool-test.cc`. (The `Makefile` has also been updated to build `thread-pool-test` and include `thread-pool.o` into the fold when building `news-aggregator`).

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.