Assignment 6: RSS News Aggregator, Take II

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 `aggregate` 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 24—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 over the course of the next seven days.

Due: Wednesday, May 24th at 11:59 p.m.

The Problem With Assignment 5

Assignment 5's `NewsAggregator` class uses `semaphores` and `mutexes` to limit the number of threads that can exist at any one time, but it **doesn't** limit the total number of threads created over the lifetime of an `aggregate` run. Even if we guarantee that no more than, say, 32 threads ever exist at any one point in time, 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 having 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 5 does that.

Here's a neat 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.

```
 pool.wait();
 cout << "All done!" << endl;
 return 0;
}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) * 10;
      sleep_for(sleepTime);
      cout << oslock << "Thread (ID: " << id << ") has finished." << endl <<
osunlock;
    });
  }

  pool.wait();
  cout << "All done!" << endl;
  return 0;
}
```

The output of the above program might look like this:

```
myth4> ./tptest
Thread (ID: 3) has started.
Thread (ID: 2) has started.
Thread (ID: 1) has started.
Thread (ID: 0) has started.
Thread (ID: 3) has finished.
Thread (ID: 4) has started.
Thread (ID: 0) has finished.
```

```
Thread (ID: 5) has started.
Thread (ID: 1) has finished.
Thread (ID: 6) has started.
Thread (ID: 4) has finished.
Thread (ID: 7) has started.
Thread (ID: 6) has finished.
Thread (ID: 8) has started.
Thread (ID: 2) has finished.
Thread (ID: 9) has started.
Thread (ID: 9) has finished.
Thread (ID: 5) has finished.
Thread (ID: 7) has finished.
Thread (ID: 8) has finished.
All done!
myth4>
```

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 solution to Assignment 5 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.

Milestone 1: Implementing the `ThreadPool` class

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++ way to classify a function that can be invoked without any arguments) to the end of a queue of such functions. Each time a function is scheduled, the dispatcher thread should be notified. Once the dispatcher has been notified, `schedule` should return right away so even *more* functions can be scheduled.

Aside: 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.

The implementation of the private `dispatcher` method should loop almost 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 repeatedly, blocking within each iteration until the dispatcher thread signals it to execute an assigned function (as described above). Once signaled, the worker should go ahead and invoke 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 scheduled functions 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.

Your `ThreadPool` implementation shouldn't orphan any memory whatsoever. We'll be analyzing your `ThreadPool` using `valgrind` to ensure no memory is leaked.

Milestone 2: Reimplementing `NewsAggregator`

Once you have a working `ThreadPool`, you should flesh out and reimplement the `NewsAggregator` class using two private `ThreadPools`. Your starter repo included pretty much the same `news-aggregator.h` and `news-aggregator.cc` files we started you off with for `assign5`.

Why two `ThreadPools` instead of just one? I want one `ThreadPool` (of size 3) to manage a collec-

tion of worker threads that download RSS XML documents, and I want a second `ThreadPool` (of size 20) to manage a second group of workers dedicated to news article downloads.

To simplify the implementation, don't worry about limiting the number of simultaneous connections to any given server. The two `ThreadPool`s you're using are pretty small, so it's unlikely any single server will be overwhelmed by your requests.

Note, however, that some Assignment 5 constraints carry over to Assignment 6.

- Make sure you never download the same exact URL twice. This is actually a constraint imposed by the XML parsing libraries themselves, and since it's easy to keep track of previously downloaded URLs, I require that you do.
- Be sure to manage the running intersections as you did for `assign5` as you come across articles with the same title and server. This'll require you to lift some code out of your Assignment 5 submission and repurpose it to contribute to Assignment 6.

Note that you should still submit code for Assignment 5 that meets the Assignment 5 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.

Relevant New Files

The files we're giving you comprise a near proper superset of the files we gave you for Assignment 5. 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`.

`tptest.cc`

`tptest.cc` contains the trivial program I posted above and is used to verify the most basic of functionality come `sanitycheck` time.

You shouldn't change this file. *If you want to exercise your `ThreadPool` implementation, you should add tests to `tpcustomtest.cc`, discussed below.*

`tpcustomtest.cc`

`tpcustomtest.cc` contains a collection of functions you can use to exercise your `ThreadPool`. This is for you to play with and flesh out with as many additional tests as you can dream of to ensure your `ThreadPool` is working brilliantly. I need you to leave `tptest.cc` alone so you can sanity check your `ThreadPool` against mine via `tptest`. That's why I've included this extra file for you.

Getting The Code

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

```
poohbear@myth30:~$ hg clone /usr/class/cs110/repos/assign6/$USER assign6
```

The code base you're cloning is more or less identical to that you cloned for Assignment 5, save for the fact that there are three more files: `thread-pool.h`, `thread-pool.cc`, `tptest.cc`, and `tpcustomtest.cc`. The `Makefile` has also been updated to build `tptest` and `tpcustomtest` and include `thread-pool.cc` into the fold when building `aggregate`.

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

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 is a solid approximation. Note that your `ThreadPool`, because it can be tested in isolation of the XML parsing library, can be examined for memory access errors and leaks.

News Aggregator, Take II Tests (100 points)

- Clean Build: 2 points
- `ThreadPool` tests: 78 points
  - Basic functionality via tests exposed in `tptest.cc` and `tpcustomtest.cc`: 40 points
  - More advanced tests that should be properly handled by a working ThreadPool: 15 points
  - Equally advanced tests that hit on an array of edge-case (but legitimate) uses cases: 15 points
  - Ensures that memory is properly managed by your `ThreadPool`—no leaks, no memory errors: 8 points
- `NewsAggregator` tests: 20 points
  - Ensure that your `ThreadPool`-backed aggregator handles a small feed list: 6 points
  - Ensure that your `ThreadPool`-backed aggregator handles a medium-sized feed list: 6 points
  - Ensure that your `ThreadPool`-backed aggregator handles a large feed list: 6 points
  - Targeted test that ensured you properly handle duplicate URLs and articles with the same title hosted by the same server: 2 points