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)>**[2], 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.

---

[2] 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 **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 completed, 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.

### Task 2: Reimplementing news-aggregator

Once you have a working **ThreadPool**, you should reimplement **news-aggregator**, using two global **ThreadPool**s. 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 a index so that **queryIndex** can return meaningful results.

Why two **ThreadPool**s 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 at most 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 URLs) 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.