Assignment 5: Helpful Hints

Here's a compilation of hints and recommendations that students have relied on in past questions to complete what is your Assignment 5:

- Because all state and functionality is managed by a C++ `NewsAggregator` class, you shouldn't need any global variables. Any state needed for aggregation can be maintained in the `private` section of the class declaration and manipulated by the class's `public` and `private` methods.
- Because your program is fully object oriented, you'll want to install `NewsAggregator` methods (as opposed to traditional functions) as your thread routines. Methods and functions are different, because the former relies on the address of the relevant object to be invisibly passed in via the `this` parameter. If you want to install a `NewsAggregator` method with signature `foobar(int number, semaphore& s)`, the method pointer can be pushed through a thread constructor in one of the three ways listed below (all of which are equivalent, with my personal preference biasing toward the second one).

```
thread t([this](int number, semaphore& s) {
  foobar(number, s);
}, n, ref(sem)); // second way

thread t(&NewsAggregator::foobar, this, n, ref(sem)); // third waythread t([this, n, &sem] { foobar(n, sem); }); // first way

thread t([this](int number, semaphore& s) {
   foobar(number, s);
}, n, ref(sem)); // second way

thread t(&NewsAggregator::foobar, this, n, ref(sem)); // third way
```

- When `semaphore`s are used to limit the number of `thread`s, it's not good enough to simply `signal` the limiting `semaphore` as the final statement of a `thread` routine, as with the code below. It's not good enough, because the surrounding `thread` may be swapped off the processor ***after*** the `signal` call but ***before*** the routine formally exits, thereby allowing a thread to still exist even though a permit allowing another thread to be created has been `signal`ed. What you really want is to schedule a call to `signal` to be made just as the `thread` is being destroyed, and that second flavor of `signal` is realize through an overloaded version of `signal`, as demonstrated below. You can learn more about the second version of `signal` and the `on_thread_exit` tag by looking at the final `myth-buster` example I posted on Monday.

```
// correct way to limit the number of threads to 6 at all times
vector<thread> threads;
semaphore permits(6);
for (size_t i = 0; i < 250; i++) {
  permits.wait();
  threads.push_back(thread([this](semaphore& s) {
    // thread safe code of your choosing
    s.signal(on_thread_exit); // schedule permits to be signaled as surrounding thread is destroyed
  }, ref(permits));
}
for (thread& t: threads) t.join();// flawed attempt to limit the number of threads to 6 at all times
vector<thread> threads;
semaphore permits(6);
for (size_t i = 0; i < 250; i++) {
   permits.wait();
   threads.push_back(thread([this](semaphore& s) {
      // thread safe code of your choosing
      s.signal();
   }, ref(permits));
}
for (thread& t: threads) t.join();

// correct way to limit the number of threads to 6 at all times
vector<thread> threads;
semaphore permits(6);
for (size_t i = 0; i < 250; i++) {
   permits.wait();
   threads.push_back(thread([this](semaphore& s) {
      // thread safe code of your choosing
```

```
      s.signal(on_thread_exit); // schedule permits to be signaled as
surrounding thread is destroyed
   }, ref(permits));
}
for (thread& t: threads) t.join();
```

- In fact, because this second version of `signal` schedules the internal `++` to happen when the `thread` exits, you can hoist the call to the top of the `thread` routine so that you know it's invoked no matter how the routine exits, as with what I present below.

```
// correct way to limit the number of threads to 6 at all times
vector<thread> threads;
semaphore permits(6);
for (size_t i = 0; i < 250; i++) {
   permits.wait();
   threads.push_back(thread([this](semaphore& s) {
      s.signal(on_thread_exit); // schedule permits to be signaled as
surrounding thread is destroyed
      // thread safe code of your choosing
   }, ref(permits));
}
for (thread& t: threads) t.join();
```

- Be careful when you share references to data with your `thread` routines. Sometimes you absolutely need to, but you need to be careful that the shared reference doesn't lead to data that unexpectedly changes or is destroyed before the `thread` routine makes use of it. You need to be sensitive to the very type of race condition that presented itself in the first version of `extroverts` we studied a little over a week ago.
- Take care to decompose and test the code of yours that manages duplicates and running intersections, because the same exact code is needed for your Assignment 6 solution. You'll more or less need to pull that code in to your own Assignment 6 repos once they're posted on Wednesday night.