Link Explorer

- You and your friends are bored so you decided to play a game where you go to a random Wikipedia page and try to find a link to another wikipedia page that is the longest (by length of the html)
  - Trust me, it’s fun!
- You decide to enlist Rust (along with the reqwest and select crates) to help you.
Sequential Link Explorer

- The most straightforward approach
- No threads => no race conditions :^)
- Let’s see how fast it is…
- (code)
Multithreaded Link Explorer

- The web requests are network bound, so we can easily overlap the wait times for these requests by running them in separate threads.
- You can see this runs considerably faster!
- Problems
  - We have this funky batching thing going on — What’s wrong with it?
  - We can easily reuse threads (really, we should be using a threadpool which you will implement in assignment 6 of CS110)
Can we do better than batching...?

- First of all, why did we need batching?
  - What happens if I just make the batch size really big...
- What’s a more effective way to limit the number of active threads/outgoing connections?
  - You saw in CS110 lecture that we can use condition variables and semaphores to impose a limit on the number of “permission slips”
  - You will see this again in Assignment 5 (News Aggregator) — as an exercise, you may wish to upgrade the link explorer example to impose limits in this way!
Lecture 10: Multithreading and Condition Variables

- The `semaphore` constructor is so short that it's inlined right in the declaration of the `semaphore` class.
- `semaphore::wait` is our generalization of `waitForPermission`.

```cpp
void semaphore::wait() {
    lock_guard<mutex> lg(m);
    cv.wait(m, [this] { return value > 0; })
    value--;
}
```

- Why does the capture clause include the `this` keyword?
  - Because the anonymous predicate function passed to `cv.wait` is just that—a regular function. Since functions aren't normally entitled to examine the `private` state of an object, the capture clause includes `this` to effectively convert the `bool`-returning function into a `bool`-returning `semaphore` method.
- `semaphore::signal` is our generalization of `grantPermission`.

```cpp
void semaphore::signal() {
    lock_guard<mutex> lg(m);
    value++;
    if (value == 1) cv.notify_all();
}
```
• Idiomatic to associate a condition variable with a mutex by putting them in a pair together and wrapping that pair in an Arc.
• We clone this pair before we move it into a thread.
  • Recall: we are NOT cloning the mutex, but rather a (reference-counted) pointer to it!
• You pass in the return value of mutex.lock().unwrap() to cv.wait(…) (or cv.wait_while(…))
• The Mutex<T> and Condvar interfaces in Rust enable us to write shorter, safer, and more legible code.
• We’ll see this in today’s live-coding example.
Semaphores can mediate access to a limited resource through giving out a limited number of “permission slips.” They can also synchronize threads to wait until a piece of data is ready (see producer/consumer) — we’ll focus on this second use case in the following example.

But they only let you increment and decrement — let’s do something more interesting.

Instead of just `sema.signal` — let’s do `sema.send (msg)`

Instead of just `sema.wait` — let’s do `sema.recv()` (which returns a msg that was previously sent)

Why might we want an abstraction like this?
Let sem = SemaPlusPlus::new();

Main Thread

Sem.recv() \rightarrow\text{comp result}

Sem.recv() \rightarrow "I'm done"

Sem.recv() \rightarrow "Something weird happened!!"

Sem.recv() \rightarrow comp result

Sem.recv() \rightarrow "I'm done"

Thread 1
Sem.send(comp_result)

Thread 2
Sem.send(Msg::ImDone)

Thread 3
Sem.send(Msg::Error)

Thread 4
Sem.send(comp_result)

Thread 5
Sem.send(Msg::ImDone)
SemaPlusPlus Implementation

- Starter code
- Finished Example