Implementing **myth-buster**!

- The **myth-buster** is a command line utility that polls all 16 **myth** machines to determine which is the least loaded.
  - By least loaded, we mean the **myth** machine that's running the fewest number of CS110 student processes.
  - Our **myth-buster** application is representative of the type of thing load balancers (e.g. `myth.stanford.edu`, `www.facebook.com`, or `www.netflix.com`) run to determine which internal server your request should forward to.
- The overall architecture of the program looks like that below. We'll present various ways to implement `compileCS110ProcessCountMap`.

```cpp
static const char *kCS110StudentIDsFile = "studentsunets.txt";
int main(int argc, char *argv[]) {
    unordered_set<string> cs110Students;
    readStudentFile(cs110Students, argv[1] != NULL ? argv[1] : kCS110StudentIDsFile);
    map<int, int> processCountMap;
    compileCS110ProcessCountMap(cs110Students, processCountMap);
    publishLeastLoadedMachineInfo(processCountMap);
    return 0;
}
```
Lecture 11: Multithreading and Networking

- Implementing **myth-buster**!

```cpp
static const char *kCS110StudentIDsFile = "studentsunets.txt";

int main(int argc, char *argv[]) {
    unordered_set<string> cs110Students;
    readStudentFile(cs110Students, argv[1] != NULL ? argv[1] : kCS110StudentIDsFile);
    map<int, int> processCountMap;
    compileCS110ProcessCountMap(cs110Students, processCountMap);
    publishLeastLoadedMachineInfo(processCountMap);
    return 0;
}
```

- **readStudentFile** updates **cs110Students** to house the SUNet IDs of all students currently enrolled in CS110. There's nothing interesting about its implementation, so I don't even show it (though you can see its implementation right here).

- **compileCS110ProcessCountMap** is more interesting, since it uses networking—our first networking example!—to poll all 16 **myths** and count CS110 student processes.

- **processCountMap** is updated to map **myth** numbers (e.g. 61) to process counts (e.g. 9).

- **publishLeastLoadedMachineInfo** traverses **processCountMap** and identifies the least loaded **myth**.
The networking details are hidden and packaged in a library routine with this prototype:

```cpp
int getNumProcesses(int num, const unordered_set<std::string>& sunetIDs);
```

- **num** is the myth number (e.g. 54 for **myth54**) and **sunetIDs** is a hashset housing the SUNet IDs of all students currently enrolled in CS110 (according to our /usr/class/cs110/repos/assign4 directory).
- Here is the sequential implementation of a `compileCS110ProcessCountMap`, which is very brute force and CS106B-ish:

```cpp
static const int kMinMythMachine = 51;
static const int kMaxMythMachine = 66;
static void compileCS110ProcessCountMap(const unordered_set<string>& sunetIDs,
                                        map<int, int>& processCountMap) {
    for (int num = kMinMythMachine; num <= kMaxMythMachine; num++) {
        int numProcesses = getNumProcesses(num, sunetIDs);
        if (numProcesses >= 0) {
            processCountMap[num] = numProcesses;
            cout << "myth" << num << " has this many CS110-student processes: " << numProcesses << 
        }
    }
}
```
Here are two sample runs of **myth-buster-sequential**, which polls each of the **myths** in sequence (i.e. without concurrency).

Each call to `getNumProcesses` is slow (about half a second), so 16 calls adds up to about 16 times that. Each of the two runs took about 5 seconds.
Lecture 11: Multithreading and Networking

- Each call to `getNumProcesses` spends most of its time off the CPU, waiting for a network connection to be established.
- Idea: poll each myth machine in its own thread of execution. By doing so, we'd align the dead times of each `getNumProcesses` call, and the total execution time will plummet.

```cpp
static void countCS110Processes(int num, const unordered_set<string>& sunetIDs, map<int, int>& processCountMap, mutex& processCountMapLock, semaphore& permits) {
  int count = getNumProcesses(num, sunetIDs);
  if (count >= 0) {
    lock_guard<mutex> lg(processCountMapLock);
    processCountMap[num] = count;
    cout << "myth" << num << " has this many CS110-student processes: " << count << endl;
  }
  permits.signal(on_thread_exit);
}

static void compileCS110ProcessCountMap(const unordered_set<string> sunetIDs, map<int, int>& processCountMap) {
  vector<thread> threads;
  mutex processCountMapLock;
  semaphore permits(8); // limit the number of threads to the number of CPUs
  for (int num = kMinMythMachine; num <= kMaxMythMachine; num++) {
    permits.wait();
    threads.push_back(thread(countCS110Processes, num, ref(sunetIDs),
                            ref(processCountMap), ref(processCountMapLock), ref(permits)));
  }
  for (thread& t: threads) t.join();
}
Lecture 11: Multithreading and Networking

- Here are key observations about the code on the prior slide:
  - Polling the myths concurrently means updating `processCountMap` concurrently. That means we need a mutex to guard access to `processCountMap`.
  - The implementation of `compileCS110ProcessCountMap` wraps a thread around each call to `getNumProcesses` while introducing a semaphore to limit the number of threads to a reasonably small number.
  - Note we use an overloaded version of `signal`. This one accepts the `on_thread_exit` tag as its only argument.
    - Rather than signaling the semaphore right there, this version schedules the signal to be sent after the entire thread routine has exited, as the thread is being destroyed.
    - That's the correct time to really signal if you're using the semaphore to track the number of active threads.
  - This new version, called `myth-buster-concurrent`, runs in about 0.75 seconds. That's a substantial improvement.
  - The full implementation of `myth-buster-concurrent` sits right here.