Announcements

- **Assignment 4**
  - Assignment 4 goes out Friday, due a week from Monday.
    - Assignment 4 is large, difficult because you need to manage the transition to C++ while coding this threads for the very first time.
    - Your Assignment 4 is being assembled out of two assignments given in prior quarters, which is why it's longer (and why you have a long time to complete it).
    - Assignments 5 and 6 will be due the day after Memorial Day and the last day of classes, respectively.
- **Midterms**
  - CAs are grading them tonight and this weekend.
  - They'll be made available next Monday.
- **Today's Agenda**
  - Implement classic **`reader-writer`** example, which uses **`semaphore`**s for cross-thread (a.k.a. rendezvous) communication.
  - Implement a sequential but very slow version of a networked program that decides which of the 30 myth machines is the least loaded. Then we'll use threading, locks, and semaphores to dramatically speed the program up without ever overloading the thread manager.

Implementing `myth-buster`

- **Core of sequential version of `myth-buster`**
  - Sequentially connects to all ~30 **`myth`** machines and asks each for the total number of processes being run by CS110 students.
  - Networking details are abstracted away and packaged in a library routine with the following prototype:

```
int getNumProcesses(unsigned short num, const unordered_set<string>& sunetIDs);
```

  - **`num`** is the myth machine number (e.g. 14 for **`myth14`**), and **`sunetIDs`** is a hashset housing the SUNet IDs of all students currently enrolled in CS110 according to our **`/usr/class/cs110/repos/assign3/`** directory.
  - Here is the sequential (and very slow) implementation of a **`compileCS110ProcessCountMap`**, which very brute force and CS106B-ish. (It assumes that **`sunetIDs`** has already been configured with the set of all CS110 student SUNet IDs, and it further assumes that **`counts`** refers to an initially empty map).
  - Full program is right here.

```
static unsigned short kMinMythMachine = 1;
static unsigned short kMaxMythMachine = 30;
static void compileCS110ProcessCountMap(const unordered_set<string>& sunetIDs,
                                        map<unsigned short, unsigned short>& counts) {
  for (unsigned short num = kMinMythMachine; num <= kMaxMythMachine; num++) {
    int numProcesses = getNumProcesses(num, sunetIDs);
    if (numProcesses >= 0) { // -1 expresses networking failure
      counts[num] = numProcesses;
      cout << "myth" << num << " has this many CS110-student processes: "
           << numProcesses << endl;
    }
  }
}
```

  - Each call to **`getNumProcesses`** is slow, and the accumulation of some 30 sequential calls is painfully slow.
  - Running the sequential version takes on the order of 80 seconds, even though 99% of that time is spent waiting for a network calls to return data.

Introduce threading

- **By introducing threading, we overlay the network stall times.**
  - Multithreaded version of the program is right here.
  - Move shared data structures and synchronization directives to global space

```
static unordered_set<string> sunetIDs;
static map<unsigned short, unsigned short> processCountMap;
static mutex processCountMapLock;
```

  - Wrap a **`thread`** around the core of the sequential code, and use a **`semaphore`** to limit the number of threads doing active work to a reasonably small number (but not so small that the program becomes sequential again) so that the thread manager doesn't get overloaded with all that many threads.
  - Note we use an overloaded version of the **`signal`** method that accepts the **`on_thread_exit`** tag as its only argument. Rather than signaling the **`semaphore`** right then and there, invoking this second version schedules the signal to be sent after the entire thread routine as exited, just as the thread is being destroyed.

```
static void countCS110Processes(unsigned short num, semaphore& s) {
  int numProcesses = getNumProcesses(num, sunetIDs);
  if (numProcesses >= 0) {
    processCountMapLock.lock();
    processCountMap[num] = numProcesses;
    processCountMapLock.unlock();
    cout << oslock << "myth" << num << " has this many CS110-student processes: "
         << numProcesses << endl << oslock;
  }

  s.signal(on_thread_exit);
}

static unsigned short kMinMythMachine = 1;
static unsigned short kMaxMythMachine = 30;
static int kMaxNumThreads = 12; // really maximum number of threads doing meaningful work
static void compileCS110ProcessCountMap() {
  vector<thread> threads;
  semaphore numAllowed(kMaxNumThreads);
  for (unsigned short num = kMinMythMachine; num <= kMaxMythMachine; num++) {
    numAllowed.wait();
    threads.push_back(thread(countCS110Processes, num, ref(numAllowed)));
  }

  for (thread& t: threads) t.join();
}
```