Announcements

- **Assignment 3 out and due on Monday, February 9th.**
  - Our next topic, threading and intraprocess concurrency, will occupy us for a while, and I know for a fact that the next assignment won't go out until the 11th at the earliest.
  - Great assignment, less work than Assignment 2, but still very tricky, so get started ASAP.
  - I'm making Assigment 2 due on Monday the 9th instead of the 11th so that you have plenty of time to review the midterm on the 12th.
- **Today's lecture**
  - Continue outlining how it is that each process can operate believing it owns the entire address space, in spite of the fact that multiple processes are running at the same time.
  - Outline how the OS — in particular, the scheduler—manages all of the currently executing processes so that each gets its fair share of the CPU.
  - Time permitting, I'll introduce the notion of a thread, which is the topic addressed by this past Friday's slide deck.

Introverts Revisited

- **C++11 provides threading and synchronization directives as part of the language now.**
  - Here's the **`introverts`** example we've seen before—this time in C++! (Full version of this program is online here).

```
#include <iostream>       // for cout, endl
#include <thread>         // for C++11 thread support
#include "ostreamlock.h"  // for CS110 iomanipulators (oslock, osunlock) used to lock down streams
using namespace std;

static void recharge() {
  cout << oslock << "I recharge by spending time alone." << endl << osunlock;
}

static const size_t kNumIntroverts = 6;
int main(int argc, char *argv[]) {
  cout << "Let's hear from " << kNumIntroverts << " introverts." << endl
  thread introverts[kNumIntroverts]; // declare array of empty thread handles
  for (thread& introvert: introverts)
    introvert = thread(recharge);    // move anonymous threads into empty handles
  for (thread& introvert: introverts)
    introvert.join();
  cout << "Everyone's recharged!" << endl;
  return 0;
}
```

Introverts Revisited

- **C++11 provides threading and synchronization directives as part of the language now.**
  - Features:
    - We declare an array of empty (e.g. non-joinable) **`thread`** handles as we did in the C version. (The **`thread`** is a class new to C++11, and the holy grail of **`thread`** documentation pages sits right here.)
    - We install the **`recharge`** function into temporary threads that are then moved (via **`operator=(thread&& other)`**) into an until-then empty **`thread`** handle.
      - This is a new form of **`operator=`** that fully transplants the contents of the **`thread`** on the right-hand side into the **`thread`** on the left-hand side, leaving the **`thread`** on the right side as fully gutted, as if it were zero-argument constructed (e.g. an empty, non-joinable **`thread`** handle).
      - This is an important clarification, because a traditional **`operator=`** would produce a second working copy of the same **`thread`**, and we don't want that. We instead want to initialize one of the **`thread`**s within the original array to be one that *now runs code*. This is how we do that.
    - The **`join`** method, not surprisingly, is equivalent to the **`pthread_join`** function we've already learned about.
    - The prototype of the thread routine—in this case, **`recharge`**—can be anything (although the return type is always ignored, so it should be **`void`** unless the same function is being used in a non-**`thread`**ed context, and a return value is useful there).
    - Big point: **`operator<<`**, unlike **`printf`**, is **not** thread-safe.
      - Even worse: a series of daisy-chained calls to **`operator<<`** is certainly not guaranteed to be executed as one, phatty atomic transaction.
      - Simple solution: I've constructed stream manipulators **`oslock`** and **`osunlock`** to lock down and subsequently release access to an **`ostream`** so that two independent threads are never trying to simultaneulously insert character data into a stream at the same time. In essence, **`oslock`** and **`osunlock`** are bookending our first *critical region*, which is a block of code that must be executed in full before any other thread can enter the same region, lest we see evidence of syncronization issues.

Thread routines can be configured in a type-safe manner

- **Thread routines can take any number of arguments.**
  - Variable argument lists—the equivalent of the ellipsis in C—are supported via a new C++11 feature called variadic templates.
  - That means we have a good amount of flexibility in how we prototype our thread routines
    - There's no question this is better than the unsafe **`void *`** funkiness that **`pthreads`** provides.
  - Here's a slightly more involed example where **`greet`** threads are configured to say hello a variable number of times. (Online version of the following program can be found right here).

```
static void greet(size_t id) {
  for (size_t i = 0; i < id; i++) {
    cout << oslock << "Greeter #" << id << " says 'Hello!'" << endl << osunlock;
    struct timespec ts = {
      0, random() % 1000000000
    };
    nanosleep(&ts, NULL);
  }
  cout << oslock << "Greeter #" << id << " has issued all of his hellos, "
       << "so he goes home!" << endl << osunlock;
}

static const size_t kNumGreeters = 6;
int main(int argc, char *argv[]) {
  srandom(time(NULL));
  cout << "Welcome to Greetland!" << endl;
  thread greeters[kNumGreeters];
  for (size_t i = 0; i < kNumGreeters; i++)
    greeters[i] = thread(greet, i + 1);
  for (thread& greeter: greeters)
    greeter.join();
  cout << "Everyone's all greeted out!" << endl;
  return 0;
}
```

Nontrivial Race Conditions

- **Multiple threads are often spawned so they can subdivide and collectively solve a single problem.**
  - Consider the case where 10 ticket agents answer telephones at United Airlines to jointly sell 1000 airline tickets (looking for full program? click here):

```
static const unsigned int kBaseIDNumber = 101;
static const unsigned int kNumAgents = 10;
static const unsigned int kNumTickets = 1000;
static mutex ticketsLock;
static unsigned int remainingTickets = kNumTickets;

static void ticketAgent(size_t id) {
  while (true) {
    ticketsLock.lock();
    if (remainingTickets == 0) break;
    remainingTickets--;
    cout << oslock << "Agent #" << id << " sold a ticket! (" << remainingTickets
         << " more to be sold)." << endl << osunlock;
    ticketsLock.unlock();
    if (shouldTakeBreak())
      takeBreak();
  }
  ticketsLock.unlock();
  cout << oslock << "Agent #" << id << " notices all tickets are sold, and goes home!"
       << endl << osunlock;
}

int main(int argc, const char *argv[]) {
  thread agents[kNumAgents];
  for (size_t i = 0; i < kNumAgents; i++)
    agents[i] = thread(ticketAgent, kBaseIDNumber + i);
  for (thread& agent: agents)
    agent.join();
  cout << "End of Business Day!" << endl;
}
```

  - Yes, we use global variables, but our defense here is that multiple threads all need access to a shared resource. **`static`** global variables are not at all uncommon in multithreaded programs (although larger programs would probably package them in a dedicated library or module).
  - There's a significant critical region in this program, and we use a **`mutex`** object to mark the beginning and end of it.
    - We'll first write this program without the **`mutex`** to illustrate what the problems are.
    - We'll then insert the **`mutex`** to understand exactly how it solves it.