Announcements

- **Today and Monday: finish up ECF, be prepared to launch into multithreading and concurrency.**
  - Assignment 2 is was due last (Thursday) night, and Assignment 3 is out today.
  - B & O Reading: Chapter 12, skipping section 12.2. (Chapter 12 is the fourth of the four chapters in the reader.)
  - Will teach you the small amount of threading using C and **`pthreads`**.
    - Textbook examples reference **`pthreads`**, which is why I want you to see at least a little bit of it in lecture before we abandon it and move over to C++.
    - Concepts taught in B & O Chapter 12 (which you are reading over the course of the next two weeks) are relevant in all languages, so don't go on thinking the reading won't be helpful.
    - I'll also illustrate the simplest of concurrency issues using **`pthreads`** so I can later argue why C++ threads, which is what we'll spend most of our time learning, make things much easier (or at least less difficult, since anything having to do with threading is very challenging).
  - Will start C++ threading once I get through three small **`pthreads`** examples.
  - Be sure to inspect code samples in **`/usr/class/cs110/lecture-examples/winter-2015/threads-c`** to see and play with working versions of everything in the coming slide decks.

What is a thread?

- **A thread is an independent thread of execution within a single process.**
  - OSes and/or programming languages allow processes to split themselves into two or more concurrently executing functions.
  - Allows for intra-process concurrency (and even true parallelism on multiprocessor and/or multicore machines)
    - Stack segment is subdivided into multiple miniature stacks.
    - Thread manager time slices and switches between simultaneously running threads in much the same way that the kernel scheduler switches between processes. (In fact, threads are often called **lightweight processes #srsly).**
    - Primary difference: threads have independent stacks, but they all share access to the same text, data, and heap segments.
      - Pro: easier to support communication between threads, because address spaces accessible are largely the same.
      - Pro: Multiple threads can access the same global data and one copy of the code.
      - Pro: One thread can share its stack space (via pointers and references) with another thread.
      - Con: No protected memory, since address space is shared. Memory errors are more common, and debugging is more difficult.
      - Con: Multiple threads can access the same global data and one copy of the code. #doubleedgedsword
      - Con: One thread can share its stack space (via pointers and references) with another thread. #mixedblessings

ANSI C doesn't provide native support for threads.

- **But POSIX threads (aka `pthreads`) comes with all standard UNIX and Linux installations of `gcc`.**
  - Primary **`pthreads`** data type is the **`pthread_t`**, which is an opaque type that helps manage how a C function is executed in its own thread of execution.
  - Only **`pthreads`** functions we'll concern ourselves with (before moving on to C++ threads) are **`pthread_create`** and **`pthread_join`**.
  - Here's a very small sample program (online version is right here):

```
#include <stdio.h>    // provides printf, which is thread-safe
#include <pthread.h>  // provides pthread_t type, thread functions

static void *recharge(void *args) {
  printf("I recharge by spending time alone.\n"); // printf is thread-safe
  return NULL;
}

static const size_t kNumIntroverts = 6;
int main(int argc, char *argv[]) {
  printf("Let's hear from %zu introverts.\n", kNumIntroverts);
  pthread_t introverts[kNumIntroverts];
  for (size_t i = 0; i < kNumIntroverts; i++)
    pthread_create(&introverts[i], NULL, recharge, NULL);
  for (size_t i = 0; i < kNumIntroverts; i++)
    pthread_join(introverts[i], NULL);
  printf("Everyone's recharged!\n");
  return 0;
}
```

  - Program above declares an array of six **`pthread_t`** *handles*.
  - Program then initializes each **`pthread_t`** (via **`pthread_create`**) by installing the **`recharge`** as the routine the each of the threads should follow while executing.
    - All thread functions must take a **`void *`** and return one as well. That's generic programming in C. #sadface
    - Second argument to **`pthread_create`** allows thread attibutes (thread priorities, etc.) to be set. We'll always pass in **`NULL`** to accept the defaults.
    - Fourth argument is passed verbatim to the thread routine once the thread is launched.
      In this case, there are no arguments, so we elect to pass in **`NULL`**.
    - Each **`recharge`** thread is eligible for processor time the instant the surrounding **`pthread_t`** is fully configured.
  - Six threads compete for thread manager's attention, and we have very little control over what choices the thread manager makes.
    - The 0^{th} thread will *probably* get processor time first.
    - Probable and guaranteed are two different words.
    - We have no official word on the permutation of possible schedules the thread manager will go with.
  - **`pthread_join`** is to threads what **`waitpid`** is to processes. The main thread blocks until the argument thread exits. (Key difference: the waiting thread need not have spawned the one it blocks on).

Race Conditions

- **Here's a slightly more involved program!**
  - Extroverts get their turn!
  - Online version is what's presented below is right here.

```
static const char *kExtroverts[] = {
    "Jon", "Pam", "Lauren", "Frank",
    "Julie", "Jack", "Tagalong Introvert"
};
static const size_t kNumExtroverts = sizeof(kExtroverts)/sizeof(kExtroverts[0]) - 1;

static void *recharge(void *args) {
  const char *name = kExtroverts[*(size_t *)args];
  printf("Hey, I'm %s.  Empowered to meet you.\n", name);
  return NULL;
}

int main() {
  printf("Let's hear from %zu extroverts.\n", kNumExtroverts);
  pthread_t extroverts[kNumExtroverts];
  for (size_t i = 0; i < kNumExtroverts; i++)
    pthread_create(&extroverts[i], NULL, recharge, &i);
  for (size_t j = 0; j < kNumExtroverts; j++)
    pthread_join(extroverts[j], NULL);
  printf("Everyone's recharged!\n");
  return 0;
}
```

  - Here's one (clearly broken) sample run:

```
myth22> ./broken-extroverts
Let's hear from 6 extroverts.
Hey, I'm Pam.  Empowered to meet you.
Hey, I'm Frank.  Empowered to meet you.
Hey, I'm Jack.  Empowered to meet you.
Hey, I'm Tagalong Introvert.  Empowered to meet you.
Hey, I'm Tagalong Introvert.  Empowered to meet you.
Hey, I'm Tagalong Introvert.  Empowered to meet you.
Everyone's recharged!
myth22>
```

Race Conditions (continued)

- **There's clearly something wrong, but what's the problem?**
  - Note that **`recharge`** references its incoming parameter this time, and that **`pthread_create`** accepts the location of the surrounding loop's index variable via its fourth argument. **`pthread_create`**'s fourth argument it always passed verbatim as the single argument to the thread routine.
  - The problem: The main thread advances **`i`** without regard for the fact that **`i`**'s address was shared with each of the six child threads.
  - At first glance, it's easy to assume that **`pthread_create`** captures not just the address of **`i`**, but the value of **`i`** itself. That assumption would be incorrect, as it copies the address and that's all.
  - The address of **`i`** (even after it goes out of scope) is constant, but its contents evolve in parallel with the execution of the six **`recharge`** threads. **`* (size_t *)args`** takes a snapshot of whatever **`i`** just happens to be at the time it's evaluated.
  - This is a simple example of what's called a *race condition*.

Race Conditions (continued)

- **Fortunately, the fix here is straightforward.**
  - Just pass the **`const char *`** itself instead.
  - Online version is what's presented below is right here.

```
static const char *kExtroverts[] = {
    "Jon", "Pam", "Lauren", "Frank",
    "Julie", "Jack", "Tagalong Introvert"
};
static const size_t kNumExtroverts = sizeof(kExtroverts)/sizeof(kExtroverts[0]) - 1;

static void *recharge(void *args) {
  const char *name = args;
  printf("Hey, I'm %s.  Empowered to meet you.\n", name);
  return NULL;
}

int main() {
  printf("Let's hear from %zu extroverts.\n", kNumExtroverts);
  pthread_t extroverts[kNumExtroverts];
  for (size_t i = 0; i < kNumExtroverts; i++)
    pthread_create(&extroverts[i], NULL, recharge, (void *) kExtroverts[i]);
  for (size_t i = 0; i < kNumExtroverts; i++)
    pthread_join(extroverts[i], NULL);
  printf("Everyone's recharged!\n");
  return 0;
}
```

Race Conditions (continued)

- **Drama averted!**
  - A different address is shared with each installation of **`recharge`**. A snapshot of the relevant C string is taken before the thread is even created, not while it's executing.
  - Race conditions are often very complicated, and avoiding them won't always be this trivial.
  - Here are a few test runs just so you see that it's fixed (and that the output varies from run to run).

```
myth22> ./extroverts
Let's hear from 6 extroverts.
Hey, I'm Jon.  Empowered to meet you.
Hey, I'm Pam.  Empowered to meet you.
Hey, I'm Julie.  Empowered to meet you.
Hey, I'm Lauren.  Empowered to meet you.
Hey, I'm Frank.  Empowered to meet you.
Hey, I'm Jack.  Empowered to meet you.
Everyone's recharged!
myth22> ./extroverts
Let's hear from 6 extroverts.
Hey, I'm Jon.  Empowered to meet you.
Hey, I'm Julie.  Empowered to meet you.
Hey, I'm Pam.  Empowered to meet you.
Hey, I'm Jack.  Empowered to meet you.
Hey, I'm Lauren.  Empowered to meet you.
Hey, I'm Frank.  Empowered to meet you.
Everyone's recharged!
myth22>
```