Agenda

- **Introduce the `semaphore`**
  - Our most recent solution to the dining philosophers problem enlisted the services of a **`mutex`** and a **`condition_variable_any`**. They collectively provide what can be best described as an integer with atomic increment, atomic decrement, and the added restriction that the integer can never go negative. Any attempt to decrement a 0 prompts that thread to block until some other thread increments it.
  - The counting variable specific to dining philosophers represents a limited resource shared among many competing threads—in essence, a limited number of permits allowing philosophers to eat.
  - We can and will generalize the idea of a counting variable by defining a **`semaphore`** class that encapsulates an integer, provides atomic increment and decrement operations via **`signal`** and **`wait`** methods, and indefinitely blocks a thread trying to decrement a **`semaphore`** surrounding a 0.
    - Conceptually, the **`semaphore`** allows us to model a shared resource—a number of remaining permission slips, a number of remaining file descriptors, a number of remaining network connections, etc—while insulating us from the complexities that come with coding with **`condition_variable_any`**. **`condition_variable_any`**s are more general than the **`semaphore`**s implemented in terms of them, but a large percentage of synchronization needs can be expressed in terms of the **`semaphore`**, which in my opinion is easier to understand.
    - Many modern languages provide native support for threading and synchronization.
      - Java, in particular, has supported threading and condition variable-style locking since the beginning. It eventually added a **`Semaphore`** class in the early 2000's.
      - Honestly, I'm not sure why C++11 decided to exclude the **`semaphore`** idea from its library set, but I think it's easier to work with than **`condition_variable_any`**s, so I'm going to introduce it so we can go forward pretending that it's just part of the C++ language.

Semaphore API

- **The semaphore API is very small.**
  - We've already mentioned that increment and potentially-blocking decrement are called **`signal`** and **`wait`**.
  - Here's the reduced interface for our own **`semaphore`** class.

```
class semaphore {
 public:
  semaphore(int value = 0);
  void wait();
  void signal();

 private:
  int value;
  std::mutex m;
  std::condition_variable_any cv;

  semaphore(const semaphore& orig) = delete;
  const semaphore& operator=(const semaphore& rhs) const = delete;
};
```

  - You can use the **`semaphore`** by including the **`semaphore.h`** file.
    - All **`Makefile`**s are configured so that you can include it and link against its implementation. You can operate like it's a C++ built-in class.
    - You can look in **`/usr/class/cs110/local/include/semaphore.h`** to confirm it's really there. ☺
  - By design, there's no **`getValue`**-like method!
    - Some **`semaphore`** designs provide such a method, but we do not.
    - Why omit it? In between the time you call it and act on it, some other thread could very well change it. Concurrency directives themselves shouldn't encourage anything that might lead to a race condition or a deadlock, and our version doesn't.
  - Notice the three private data members are akin to the four global variables we introduced to limit the number of philosophers grabbing forks.
  - I remove the copy constructor and assignment operator (using the **`delete`** keyword), because neither the **`mutex`** nor **`condition_variable_any`** are copy constructable, copy assignable, or even movable.
    - In a nutshell, this means you need to pass all instances of **`mutex`**, **`condition_variable_any`**, and **`semaphore`** around by reference or via its address.

Semaphore Implementation

- **The implementation is very short and very dense.**
  - Here's the implementation of the constructor:

```
semaphore::semaphore(int value) : value(value), {}
```

  - `m` and `cv` are zero-argument constructed.
  - Data members are constructed in the order they appear in the `class` definition (not necessarily the order they appear in the initialization list).
  - Here are the implementations of `wait` and `signal`, which look like our most recent implementations of `waitForPermission` and `grantPermission`:

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

void semaphore::signal() {
  lock_guard<mutex> lg(m);
  value++;
  if (value == 1) cv.notify_all();
}
```

  - Note that `this` needs to be captured by the on-the-fly predicate function we pass to `cv.wait`. We need access to the `value` data member, and capturing the address of the surrounding object allows this.
    - `[&value]` works with `g++`, but it's off C++11 specification and won't necessarily work with other compilers.

Final Version of Dining Philosophers

- **Using `semaphores`, in my opinion, improves the narrative.**
  - Strip out exposed **`int`**, **`mutex`**, and **`condition_variable_any`** and replace with single **`semaphore`**.
  - No longer need separate **`waitForPermission`** and **`grantPermission`** functions. #like
  - One last time, with feeling:

```
static mutex forks[kNumForks];
static semaphore numAllowed(kNumForks - 1);

static void think(unsigned int id) {
  cout << oslock << id << " starts thinking." << endl << osunlock;
  sleep_for(getThinkDuration());
  cout << oslock << id << " all done thinking. " << endl << osunlock;
}

static void eat(unsigned int id) {
  unsigned int left = id;
  unsigned int right = (id + 1) % kNumForks;
  numAllowed.wait(); // atomic -- that blocks on attempt to decrement 0
  forks[left].lock();
  forks[right].lock();
  cout << oslock << id << " starts eating om nom nom nom." << endl << osunlock;
  sleep_for(getEatDuration());
  cout << oslock << id << " all done eating." << endl << osunlock;
  numAllowed.signal(); // atomic ++, never blocks, possibly unblocks other waiting threads
  forks[left].unlock();
  forks[right].unlock();
}
```

  - Parting comments:
    - It's easy to understand the transactional **`++`** and **`--`** that comes with **`signal`** and **`wait`**.
    - The thread yield that comes with a **`wait`** on a **`semaphore`** value of 0 is more difficult to understand. Given that a **`semaphore`** represents a shared, limited resource, blocking and doing nothing until that resource becomes available is almost always the right thing to do.
    - Make sure you understand the many pros on this approach over the busy waiting approach we initially used to avert the threat of deadlock.
    - Can you think of any situations when busy waiting (also called spin locking) might be the right approach? #thoughtquestion

Canonical Reader/Writer Example

- **Thread Rendezvous**
  - **`semaphore::wait()`** and **`semaphore::signal()`** can be exploited to provide a *different* form of thread communication: *rendezvous*.
  - Here's our first example:

```
static const unsigned int kNumBuffers = 30;
static const unsigned int kNumCycles = 4;

static char buffer[kNumBuffers];
static semaphore emptyBuffers(kNumBuffers);
static semaphore fullBuffers(0);

static void writer() {
  cout << oslock << "Writer: ready to write." << endl << osunlock;
  for (unsigned int i = 0; i < kNumCycles * kNumBuffers; i++) {
    char ch = prepareData();
    emptyBuffers.wait();   // don't try to write to a slot unless you know it's empty
    buffer[i % kNumBuffers] = ch;
    fullBuffers.signal();  // signal reader there's more stuff to read
    cout << oslock << "Writer: published data packet with character '"
         << ch << "'." << endl << osunlock;
  }
}

static void reader() {
  cout << oslock << "\t\tReader: ready to read." << endl << osunlock;
  for (unsigned int i = 0; i < kNumCycles * kNumBuffers; i++) {
    fullBuffers.wait();    // don't try to read from a slot unless you know it's full
    char ch = buffer[i % kNumBuffers];
    emptyBuffers.signal(); // signal writer there's a slot that can receive data
    processData(ch);
    cout << oslock << "\t\tReader: consumed data packet "
         << "with character '" << ch << "'." << endl << osunlock;
  }
}

int main(int argc, const char *argv[]) {
  thread w(writer);
  thread r(reader);
  w.join();
  r.join();
  return 0;
}
```

  - Think of the **`writer`** thread as one that **serves** data to a network connection, and think of the **`reader`** thread as one that **consumes** it.
  - Two **`semaphore`**s are used synchronize the two so that:
    - **`reader`** is never further along than **`writer`**, and
    - **`writer`** is never so far ahead of **`reader`** that it clobbers data that has yet to be consumed.