The Dining Philosophers Problem, continued
○ The last slide deck left you with the three implementations for `waitForPermission` and `grantPermission`:

```cpp
static void waitForPermission(size_t& permits, condition_variable_any& cv, mutex& m) {
    lock_guard<mutex> lg(m);
    while (permits == 0) cv.wait(m);
    permits--;
}

static void grantPermission(size_t& permits, condition_variable_any& cv, mutex& m) {
    lock_guard<mutex> lg(m);
    permits++;
    if (permits == 1) cv.notify_all();
}
```

○ It's worth noting that `waitForPermission` requires a `while` loop instead of an `if` test.
  ■ It's possible the permit that just became available is immediately consumed by the thread that just returned it. Yes, it's unlikely. But it's technically possible, since we don't control how threads are scheduled.
The Dining Philosophers Problem, continued

- While loops around `cv.wait(m)` calls are so common that the `condition_variable_any` class exports a two-argument version of `wait` whose implementation is a `while` loop around the first. That second version looks like this:

```cpp
template <Predicate pred>
void condition_variable_any::wait(mutex& m, Pred pred) {
    while (!pred()) wait(m);
}
```

- It's a template method, because the second argument supplied via `pred` can be anything capable of standing in for a zero-argument, `bool`-returning function.
- The first `waitForPermissions` can be rewritten to rely on this new version, as with:

```cpp
static void waitForPermission(size_t& permits, condition_variable_any& cv, mutex& m) {
    lock_guard<mutex> lg(m);
    cv.wait(m, [&permits] { return permits > 0; });
    permits--;
}
```
Fundamentally, the `size_t`, `condition_variable_any`, and `mutex` are collectively working together to track a number of resources—in this case, four permission slips.

- They provide thread-safe increment in `grantPermission` and thread-safe decrement in `waitForPermission`.
- They work to ensure that a thread blocked on zero permission slips goes to sleep indefinitely, and that it remains asleep until another thread returns one.

In our latest `dining-philosopher` example, we relied on these three variables to collectively manage a thread-safe accounting of four permission slips. However!

- There is little about the implementation that requires the original number be four. Had we gone with 20 philosophers and and 19 permission slips, `waitForPermission` and `grantPermission` would still work as is.
- The idea of maintaining a thread-safe, generalized counter is so useful that most programming languages include more generic support for it. That support normally comes under the name of a `semaphore`.
- For reason that aren't entirely clear to me, standard C++ omits the `semaphore` from its standard libraries. My guess as to why? It's easily built in terms of other supported constructs, so it was deemed unnecessary to provide official support for it.
Because it's so useful, we'll generalize the idea of a thread-safe counting variable by defining our own `semaphore` class, and then pretend that it's part of the C++ language.

`condition_variable_any`s are more general than the `semaphores` 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.

Here's the compressed interface for our `semaphore` class:

```cpp
class semaphore {
public:
    semaphore(int value = 0): value(value) {} // thread-safe decrement, with block on decrement against 0
    void wait(); // thread-safe increment, notify any blocked threads
    void signal();
private:
    int value;
    std::mutex m;
    std::condition_variable_any cv;
};
```
Lecture 16: Multithreading and Semaphores

- The `semaphore` constructor is so short that it's inlined right in the declaration.
- `semaphore::wait` is our generalization of `waitForPermission`.

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

- The anonymous predicate function passed to `cv.wait` is just that—a regular function. Because functions aren't normally entitled to examine the `private` state of an object, the capture clause includes `this` to effectively convert the `bool`-returning function into a `bool`-returning `semaphore` method.
- `semaphore::signal` is our generalization of `grantPermission`.

```cpp
void semaphore::signal() {
  lock_guard<mutex> lg(m);
  value++;
  if (value == 1) cv.notify_all();
}
```
Lecture 16: Multithreading and Semaphores

- Here's our final version of the **dining-philosophers**.
  - It strips out the exposed `size_t`, `mutex`, and `condition_variable_any` and replaces with a single `semaphore`.
  - It updates the thread constructors to pass a single reference to that `semaphore`.

```c
static void philosopher(size_t id, mutex& left, mutex& right, semaphore& permits) {
    for (size_t i = 0; i < 3; i++) {
        think(id);
        eat(id, left, right, permits);
    }
}

int main(int argc, const char *argv[]) {
    semaphore permits(4);
    mutex forks[5];
    thread philosophers[5];
    for (size_t i = 0; i < 5; i++) {
        mutex& left = forks[i], & right = forks[(i + 1) % 5];
        philosophers[i] = thread(philosopher, i, ref(left), ref(right), ref(permits));
    }
    for (thread& p: philosophers) p.join();
    return 0;
}
```
Lecture 16: Multithreading and Semaphores

- **eat** now relies on that **semaphore** to play the role previously played by **waitForPermission** and **grantPermission**.

```cpp
static void eat(size_t id, mutex& left, mutex& right, semaphore& permits) {
    permits.wait();
    left.lock();
    right.lock();
    cout << oslock << id << " starts eating om nom nom nom."
    << endl << osunlock;
    sleep_for(getEatTime());
    cout << oslock << id << " all done eating."
    << endl << osunlock;
    permits.signal();
    left.unlock();
    right.unlock();
}
```

- We could switch the order of the last two lines, so that **right.unlock()** precedes **left.unlock()**. Is the switch a good idea? A bad one? Or is it really just arbitrary?
- One student suggested we use a **mutex** to bundle the calls to **left.lock()** and **right.lock()** into a critical region. Is this another solution to the deadlock problem?
- We could lift the **permits.signal()** call up to appear in between **right.lock()** and the first **cout** statement. Is that valid? Why or why not?
Thread rendezvous using the `semaphore`
- `semaphore::wait` and `semaphore::signal` can be leveraged to support a different form of communication: **thread rendezvous**.
- Thread rendezvous is a generalization of `thread::join`. It allows one thread to stall—via `semaphore::wait`—until another thread calls `semaphore::signal`, often because the signaling thread just prepared some data that the waiting thread needs before it can continue.

To illustrate when thread rendezvous is useful, we'll implement a simple program without it, and see how thread rendezvous can be used to repair some of its problems.
- The program has two meaningful threads of execution: one thread publishes content to a shared buffer, and a second reads that content as it becomes available.
- The program is a nod to the communication in place between a web server and a browser. The server publishes content over a dedicated communication channel, and the browser consumes that content.
- The program also reminds me of how two independent processes behave when one writes to a pipe, a second reads from it, and how the write and read processes behave when the pipe is full (in principle, a possibility) or empty.
Consider the following program, where concurrency directives have been intentionally omitted. (The full program is right here.)

```cpp
static void writer(char buffer[]) {
    cout << oslock << "Writer: ready to write." << endl << osunlock;
    for (size_t i = 0; i < 320; i++) { // 320 is 40 cycles around the circular buffer of length 8
        char ch = prepareData();
        buffer[i % 8] = ch;
        cout << oslock << "Writer: published data packet with character '" << ch << "." << endl << osunlock;
    }
}

static void reader(char buffer[]) {
    cout << oslock << "Reader: ready to read." << endl << osunlock;
    for (size_t i = 0; i < 320; i++) { // 320 is 40 cycles around the circular buffer of length 8
        char ch = buffer[i % 8];
        processData(ch);
        cout << oslock << "Reader: consumed data packet " << "with character '" << ch << "." << endl << osunlock;
    }
}

int main(int argc, const char *argv[]) {
    char buffer[8];
    thread w(writer, buffer);
    thread r(reader, buffer);
    w.join();
    r.join();
    return 0;
}
```
Lecture 16: Multithreading and Semaphores

- Here's what works:
  - Because the main thread declares a circular buffer and shares it with both children, the children each agree where content is stored.
    - Think of the buffer as the state maintained by the implementation of pipe, or the state maintained by an internet connection between a server and a client.
  - The writer thread publishes content to the circular buffer, and the reader thread consumes that same content as it's written. Each thread cycles through the buffer the same number of times, and they both agree that \( i \% 8 \) identifies the next slot of interest.

- Here's what's broken:
  - Each thread runs more or less independently of the other, without consulting the other to see how much progress it's made.
  - In particular, there's nothing in place to inform the reader that the slot it wants to read from has meaningful data in it. It's possible the writer just hasn't gotten that far yet.
  - Similarly, there's nothing preventing the writer from advancing so far ahead that it begins to overwrite content that has yet to be consumed by the reader.
Lecture 16: Multithreading and Semaphores

- One solution? Maintain two semaphores.
  - One can track the number of slots that can be written to without clobbering yet-to-be-consumed data. We'll call it `emptyBuffers`, and we'll initialize it to 8.
  - A second can track the number of slots that contain yet-to-be-consumed data that can be safely read. We'll call it `fullBuffers`, and we'll initialize it to 0.
- Here's the new `main` program that declares, initializes, and shares the two semaphores.

```c
int main(int argc, const char *argv[]) {
  char buffer[8];
  semaphore fullBuffers, emptyBuffers(8);
  thread w(writer, buffer, ref(fullBuffers), ref(emptyBuffers));
  thread r(reader, buffer, ref(fullBuffers), ref(emptyBuffers));
  w.join();
  r.join();
  return 0;
}
```

- The writer thread waits until at least one buffer is empty before writing. Once it writes, it'll increment the full buffer count by one.
- The reader thread waits until at least one buffer is full before reading. Once it reads, it increments the empty buffer count by one.
Here are the two new thread routines:

```cpp
static void writer(char buffer[], semaphore& full, semaphore& empty) {
    cout << oslock << "Writer: ready to write." << endl << osunlock;
    for (size_t i = 0; i < 320; i++) {
        // 320 is 40 cycles around the circular buffer of length 8
        char ch = prepareData();
        empty.wait(); // don't try to write to a slot unless you know it's empty
        buffer[i % 8] = ch;
        full.signal(); // signal reader there's more stuff to read
        cout << oslock << "Writer: published data packet with character '" << ch << ")." << endl << osunlock;
    }
}

static void reader(char buffer[], semaphore& full, semaphore& empty) {
    cout << oslock << "\t\tReader: ready to read." << endl << osunlock;
    for (size_t i = 0; i < 320; i++) {
        // 320 is 40 cycles around the circular buffer of length 8
        full.wait(); // don't try to read from a slot unless you know it's full
        char ch = buffer[i % 8];
        empty.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;
    }
}
```

The reader and writer rely on these **semaphores** to inform the other how much work they can do before being necessarily forced off the CPU.

Thought question: can we rely on just one **semaphore** instead of two? Why or why not?