Lecture 8

Note: Reading these lecture notes is not a substitute for watching the lecture. I frequently go off script, and you are responsible for understanding everything I talk about in lecture unless I specify otherwise.

Revisiting `workers.cc` "primitive thread pool" example

We're going to step by `workers.cc` to revisit the synchronization involved. See the slides for this lecture. Make sure you understand what mutexes and condition variables do – ask questions early!

Semaphores

I don't know why this primitive ended up being called a semaphore, but that is the name we have! (A semaphore is a visual signaling mechanism, e.g. railway semaphore or flag semaphore.) I think a semaphore is best described by analogy:

A semaphore is like a bucket of balls. A thread can put balls in a bucket at will. If a thread needs one of the balls in order to do something, it'll take it from the bucket; if the bucket is empty, it will wait for a ball to be added. (Another common analogy is a bucket of permission slips. A thread can add permission slips to the bucket, and another can wait for them to be added.)

A `semaphore` object has two methods:

- `signal()`
  - Adds a ball to the bucket.
  - **Never blocks.**
- `wait()`
  - If a ball is in the bucket, the thread takes the ball and returns immediately.
  - If no ball is in the bucket, waits until one is available, then takes the ball and returns.

This setup might sound familiar. In fact, we already saw this pattern in `workers.cc`: the scheduler adds something to the queue (adding a ball to the bucket), and the workers wait to pop something from the queue (i.e. waiting to take a ball from the bucket). As it turns out, this pattern is *extremely* common in multithreading, and that's why the semaphore is such a common synchronization primitive. Flip back and forth between the last two slides for today's lecture to see how the code gets simplified when we use a semaphore instead of mutexes + condition variables.

With mutex/condition variable:

```
size_t numQueued = 0;
mutex numQueuedLock;
conditional_variable_any queueCv;

static void runWorker(size_t id) {
    while (true) {
        numQueuedLock.lock();

        queueCv.wait(numQueuedLock, [&](){return numQueued > 0});

        // Pop from queue, and do expensive processing
        numQueued--;
        cout << oslock << "Worker #" << id << ": popped from queue."
            << endl << osunlock;
        numQueuedLock.unlock();
        sleep_for(1500);
    }
}

static void runScheduler() {
    for (size_t i = 0; i < 10; i++) {
        sleep_for(300);
        lock_guard<mutex> lg(numQueuedLock);
        numQueued++;
        queueCv.notify_all();
        cout << oslock << "Scheduler: added to queue."
            << endl << osunlock;
    }
}
```

With a semaphore:

```
semaphore sem;

static void runWorker(size_t id) {
    while (true) {
        sem.wait();
        cout << oslock << "Worker #" << id << ": popped from queue."
            << endl << osunlock;
        sleep_for(1500);
    }
}

static void runScheduler() {
    for (size_t i = 0; i < 10; i++) {
        sleep_for(300);
        sem.signal();
        cout << oslock << "Scheduler: added to queue."
            << endl << osunlock;
    }
}
```

See how much simpler the code is with a semaphore! Takeaways:

- **Anything you can do with a semaphore, you can also do with a condition variable.** After all, we *built* the semaphore using a condition variable.
- **If you can build it using a semaphore, build it using a semaphore.** Sometimes, you may need to wait for complicated conditions, and a semaphore won't suffice. Use a condition variable for those cases. But if you can, use a semaphore; it's so much simpler!

Implementing semaphores

Most programming languages include semaphores out of the box. For some reason, C++ doesn't include a semaphore primitive. We are providing one for your use in this class. The header file is in `/usr/class/cs110/local/include/semaphore.h`, and the source code is in `/usr/class/cs110/local/src/threads/semaphore.cc`. A simplified version reads like this:

```
class semaphore {
```

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

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

    /* We disable the copy operators. It's almost always a mistake to make a
     * copy of a semaphore; if you have a semaphore with 1 ball in the bucket
     * and two threads make independent copies, they could both take a ball,
     * despite there only being 1 ball. */
    semaphore(const semaphore& orig) = delete;
    const semaphore& operator=(const semaphore& rhs) const = delete;
};

/* The constructor allows us to start off with some preexisting number of balls
 * in the bucket. */
semaphore::semaphore(int value) : value(value) {}

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();
}
```

Practice: Avoiding deadlock with semaphores

There are 5 philosophers sitting around a table with an enormous plate of spaghetti on it. There is a fork in between each philosopher (for a total of 5 forks). Each philosopher thinks for some time, then grabs both forks surrounding him and starts eating, then puts them back and goes back to philosophizing.

We can represent this using the following code:

```
static void philosophize(size_t id) {
    for (size_t i = 0; i < 3; i++) {
        think(id);
        eat(id);
    }
}

// Each mutex in this array represents a fork. If the fork is being used by
// someone, the lock is locked; if it's not in use, the lock is up for grabs.
static mutex forks[5];
static void eat(size_t id) {
    size_t left = id;
    size_t right = (id + 1) % 5;
    forks[left].lock();
    forks[right].lock();
    sleep_for(randomEatTime());
    forks[left].unlock();
    forks[right].unlock();
}
```

This code suffers from *deadlock*. Consider this possibility: At the very same time, every philosopher grabs the fork to the left of him. Then, when each philosopher goes to grab the fork to his right, it will have already been taken by the philosopher to his right.

We can avoid this by only allowing up to 4 philosophers to eat at a time. We can create a semaphore that is initialized with 4 “permission slips in the bucket.” For each philosopher to start eating, he first has to grab a permission slip. When he’s done, he puts it back in the bucket for someone else to grab.

```
static void philosophize(size_t id) {
    for (size_t i = 0; i < 3; i++) {
        think(id);
        eat(id);
    }
}

// Each mutex in this array represents a fork. If the fork is being used by
// someone, the lock is locked; if it's not in use, the lock is up for grabs.
static mutex forks[5];
static semaphore sem;
static void eat(size_t id) {
    sem.wait();
    size_t left = id;
    size_t right = (id + 1) % 5;
    forks[left].lock();
    forks[right].lock();
    sleep_for(randomEatTime());
    forks[left].unlock();
    forks[right].unlock();
    sem.signal();
}
```

Practice: Communicating using a ring buffer

We can use semaphores to implement communication between two threads using a ring buffer. One thread reads from the ring buffer, but it needs to ensure the writer has written data before it attempts to read it – one semaphore will be used for this. Equally important, however, is this: the writer needs to ensure that the reader has read information before it loops to the beginning of the ring buffer and overwrites those bytes. A second semaphore is used for this.

```
static char buffer[10];
static semaphore dataRead(8);
static semaphore dataWritten(0);

static void writer() {
    for (size_t i = 0; i < 104; i++) {
        char ch = nextChar();
        // Make sure the reader thread has read bytes before we overwrite them
        dataRead.wait();
        buffer[i % sizeof(buffer)] = ch;
        cout << oslock << "Writer: published character '" << ch << "'" << endl
```

```
            << osunlock;
        // Signal the reader that there's stuff to read
        dataWritten.signal();
    }
}

static void reader() {
    for (size_t i = 0; i < 104; i++) {
        // Wait for the writer to send us something
        dataWritten.wait();
        char ch = buffer[i % sizeof(buffer)];
        cout << oslock << "\t\t\t\tReader: read character '" << ch << "'" << endl
            << osunlock;
        // Tell the writer that it's okay to send more data
        dataRead.signal();
        // Do some expensive processing of the data
        randomSleep();
    }
}

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