Avery's notes for advanced multithreading. Not guaranteed to be bug-free.

Review from Last Time

Our Model:

N producers putting tasks (that we'll number with ints) into a bounded queue, and M consumers taking tasks out of the queue.

Example Applications

- Students putting requests into the LaIR queue, section leaders servicing requests and taking them out of the LaIR queue.
- Your user programs on your computer sends tasks to the operating system, which determines which processor your code should be run on.
- Your browser sends HTTP requests to a website, and servers for that website service these requests.

Lambdas

BTW, you can use lambdas instead of functions.

Bounded Buffer Example

Problem: Race condition between consumers and producers. When does the consumer know that the producer has put stuff into the queue? What should the consumer do in the meantime?
Solution for now: busy-wait. Spin continually in a loop until count != 0 (there is an element).

Problem: Race condition between consumers. All three consumers try to service request 0.
Solution: Use a mutex to protect the critical region, which is the lines adding to the queue (protects 'count', 'buffer', 'rear', 'front').

Problem: busy-waiting is wasteful.
Solution: use a condition variable.

```
cv.wait(mutex, predicate)
```

equivalent to

```
while (!predicate()) {
      mutex.unlock();
      // sleep, until prompted by another thread
      mutex.lock(); // necessary to protect the predicate (which uses 'count')
}
```

Problem: we have deadlock, all consumers don't do anything!
Solution: notice above the consumers sleep until prompted by another thread. Specifically we want to wake the consumers when a producer has added to the queue. Call `cv.notify_one()`, which will wake up one of the consumers
We can do similar things with the producers.

Final solution:

```
// run by producers
void deposit(int data) {
	{
		// ensure no other threads are editing the buffer
		std::unique_lock<std::mutex> l(lock);

		// if full, wait until a consumer tells me it's not full any more
		not_full.wait(l, [this](){return count != capacity; });

		// add to the queue
		buffer[rear] = data;
		rear = (rear + 1) % capacity;
		++count;

	} // mutex is unlocked here

	// tell consumers we just added a task
	not_empty.notify_one();
}

// run by consumers
int fetch() {
	{
		// ensure no other threads are editing the buffer
		std::unique_lock<std::mutex> l(lock);

		// if empty, wait until a producer tells me it's not empty any more
		not_empty.wait(l, [this](){return count != 0; });

		// remove from the queue
		int result = buffer[front];
		front = (front + 1) % capacity;
		--count;
	} // mutex is unlocked here

	// tell producer we just freed up a space
	not_full.notify_one();

	return result;
}
```

Conclusion:

thread creation: constructor takes in a function to be run
followed by the parameters to be passed in.
std::ref converts a variable into a reference.

mutex: protect regions that cannot be processed concurrently.

lock_guard/unique_mutex: RAII-compliant wrapper for the mutex
note the example of a custom scope with the braces above.

condition variables: for cross-thread communication
wait function sleeps without busy-waiting, until pre-empted by
a notify_one or notify_all call by another thread
then it wakes up, locks the mutex, and checks the condition.
if condition is not met, then goes back to sleep.

More to read: semaphores (added in C++20!), atomics library.
Classes to take: CS 110

Bottom Line: multithreading is very powerful, but very hard to get right. This lecture was just to show you that multithreading isn't easy, and there are a lot of problems that programmers face. The modern C++ library has provided us with powerful synchronization primitives. We don't expect you to write a multithreaded application just from sitting in this lecture (yep, we scraped part 9 from assignment 3!) Take CS 110 to learn more.

By the way, this class is actually super helpful for CS 110. CS 110 students are actually expected to figure out around a third of the material in this class (particularly lambdas, algorithms, copy and move, smart pointers) on their own throughout the quarter. You're all super well-prepared.

Hope you enjoyed CS 106L! :)