ECE 3574: Applied Software Design

Thread Safe Queue

Today we are going to look in detail at how to make a data structure thread-safe.

- Review of std::queue
- push
- empty
- try_pop
- wait_and_pop
- Message Queues
- Exercise

Review `std::queue`

A first-in-first-out queue with (basic) methods

- push
- pop
- empty

Like all standard containers, `std::queue` is not thread-safe

- Q: How can we adapt the queue to protect access?
- A: mutexes and condition variables

We protect each method with a mutex.

The interface

```
template<typename T>
class ThreadSafeQueue
{
 public:

  void push(const T & value);

  bool empty() const;

  bool try_pop(T& popped_value);

  void wait_and_pop(T& popped_value);

 private:
  std::queue<T> the_queue;
  mutable std::mutex the_mutex;
  std::condition_variable the_condition_variable;
};
```

Simplest case: empty member function

```
template<typename T>
bool ThreadSafeQueue<T>::empty() const {
    std::lock_guard<std::mutex> lock(the_mutex);
    return the_queue.empty();
}
```

push member function

```
template<typename T>
void ThreadSafeQueue<T>::push(const T& value) {
    std::unique_lock<std::mutex> lock(the_mutex);
    the_queue.push(value);
    lock.unlock();
    the_condition_variable.notify_one();
  }
```

try_pop member function

No waiting, returns true on success, popped value as an output argument.

```
template<typename T>
bool ThreadSafeQueue<T>::try_pop(T &popped_value) {
    std::lock_guard<std::mutex> lock(the_mutex);
    if (the_queue.empty()) {
      return false;
    }

    popped_value = the_queue.front();
    the_queue.pop();
    return true;
  }
```

wait_and_pop member function

Wait for available, returns popped value as an output argument.

```
template<typename T>
void ThreadSafeQueue<T>::wait_and_pop(T &popped_value) {
    std::unique_lock<std::mutex> lock(the_mutex);
    while (the_queue.empty()) {
      the_condition_variable.wait(lock);
    }

    popped_value = the_queue.front();
    the_queue.pop();
  }
```

Thread-safe queues are a good way to implement message passing between threads, where they are called *Message Queues*.

- Each thread has a pointer or reference to a shared *input* ThreadSafeQueue holding units of work
- Each thread has a pointer or reference to a shared *output* ThreadSafeQueue holding results of work
- Each thread calls `wait_and_pop` on input queue, does the work, then calls `push` on the output queue

Often a single thread, the *Producer*, pushes into the input queue and pops from the output queue. The other threads act as *Workers* or *Consumers*.

Exercise

See the website.

Next Actions and Reminders

- Read about Producer/Consumer Pattern