Outline

- Advanced locking with mutexes
- std::atomic
- Thread synchronization

Advanced locking with mutexes

- Recursive locking:
  - **`std::recursive_mutex`**
  - Enables the same thread to lock the mutex twice
  - Useful in the context of recursive functions

Advanced locking with mutexes

- Timed locking:
  - **`std::timed_mutex`**
  - **`std::timed_recursive_mutex`**
  - Similar to **`std::mutex`**, but enables a thread to do something else while waiting for another thread to **`unlock`**
  - Additional functionality:

    `try_lock` : tries to lock, returns false if failed
    `try_lock_for` : tries to lock for specified time
    `try_lock_until` : tries to lock until specified time

std::call_once

- It is possible that some operations are to be done only once
- Use

```
std::call_once( std::once_flag, function);
```

std::call_once Example

```
#include <iostream>
#include <thread>
#include <mutex>

std::once_flag flag1;
void printHello() {  std::cout << "Hello\n"; }
void threadFunction(){
    std::call_once(flag1, printHello);
}

int main(){
    std::thread st1(threadFunction);
    std::thread st2(threadFunction);
    std::thread st3(threadFunction);
    st1.join();
    st2.join();
    st3.join();
    return 0;
}
```

Conditional variable

Unique call

Outline

- Advanced locking with mutexes
- std::atomic
- Thread synchronization

std::atomic

- C++11 introduces atomic types as a generic template class that can be wrapped around any type

```
std::atomic<Type> object;
```

- Can be used with any type
- Makes the operations on that type atomic
- Locking technique depends on type, and can be very fast for small objects (faster than mutex!)

Example atomic

- Back to our counter example

```
class Counter {
public:
  Counter() { m_value = 0; };

  int getValue() { return m_value; };
  void increment() {
    m_mutex.lock();

    ++m_value;
    m_mutex.unlock();
  };


private:
  int m_value;

  std::mutex m_mutex;
};
```

Example atomic

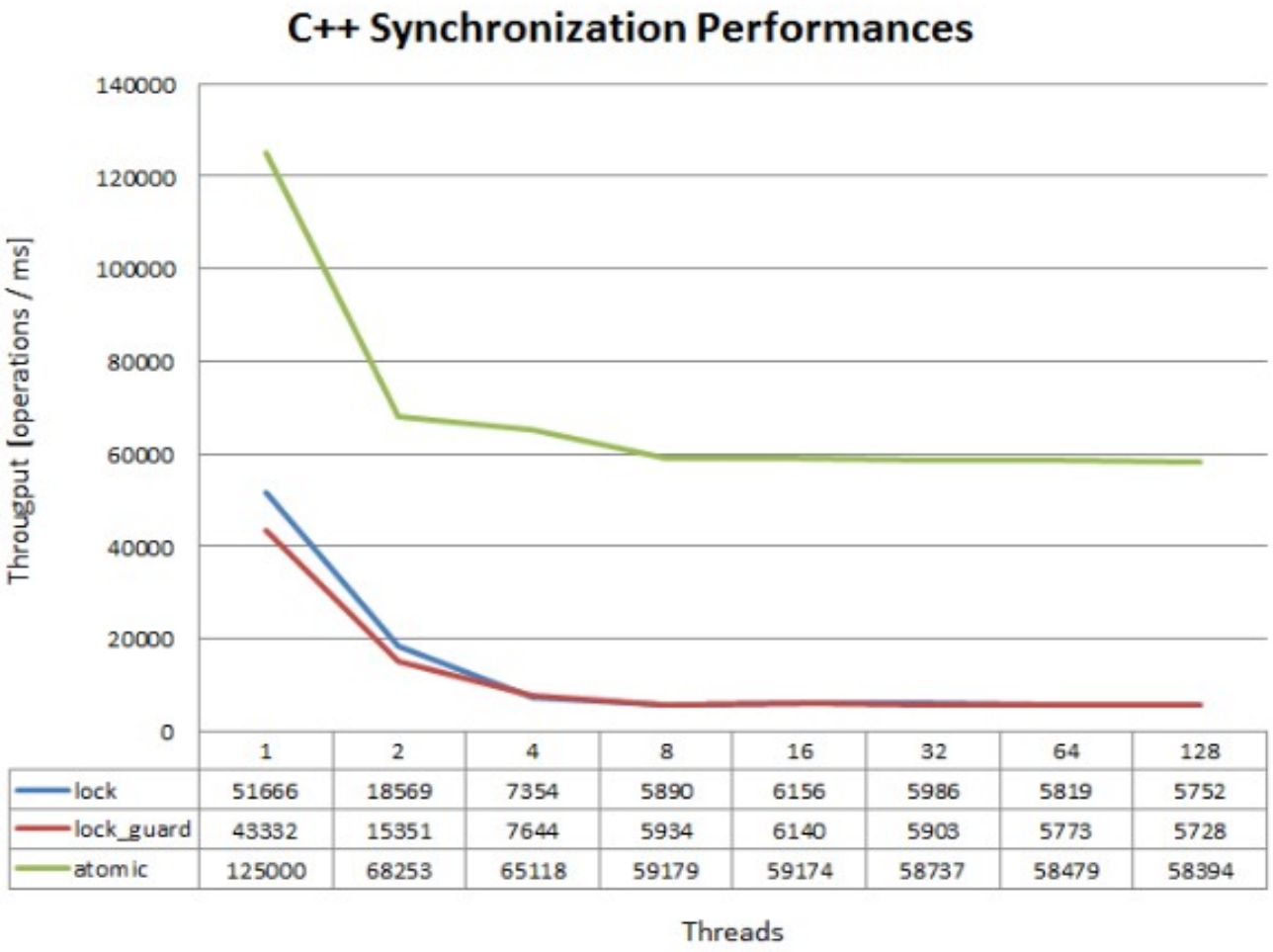
- Back to our counter example

```
class Counter {
public:
  Counter() { m_value = 0; };

  int getValue() { return m_value; };
  void increment() {
    ++m_value;
  };

private:
  std::atomic<int> m_value;
};
```

Speed comparison



	1	2	4	8	16	32	64	128
lock	51666	18569	7354	5890	6156	5986	5819	5752
lock_guard	43332	15351	7644	5934	6140	5903	5773	5728
atomic	125000	68253	65118	59179	59174	58737	58479	58394

Outline

- Advanced locking with mutexes
- std::atomic
- Thread synchronization

Synchronization between threads

- Apart from just protecting data, sometimes we may wish for one thread to wait until another thread has something done
- In C++:
  - Conditional variables
  - Futures

std::condition_variable

- A synchronization primitive that can be used to block a thread or multiple threads at the same time, until
  - A notification is received from another thread
  - A time-out expires

std::condition_variable

- A thread that intends to wait on std::condition_variable has to acquire a std::unique_lock first
- The wait operations atomically release the mutex and suspend the execution of the thread
- When the condition variable is notified, the thread is awakened, and the mutex is reacquired

Example

```
std::mutex mut;
std::queue<data_chunk> data_queue;
std::condition_variable data_cond;

void data_preparation_thread() {
  while( more_data_to_prepare() ) {
    data_chunk data = prepare_data();
    std::lock_guard<std::mutex> lk(mut);
    data_queue.push(data);
    data_cond.notify_one();
  }
}

void data_processing_thread() {
  while(true) {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[]{return !data_queue.empty();});
    data_chunk data = data_queue.front();
    data_queue.pop();
    lk.unlock();
    process(data);
    if(is_last_chunk(data))
      break;
  }
}
```

Mutex to protect resource

Example

```
std::mutex mut;
std::queue<data_chunk> data_queue;
std::condition_variable data_cond;

void data_preparation_thread() {
  while( more_data_to_prepare() ) {
    data_chunk data = prepare_data();
    std::lock_guard<std::mutex> lk(mut);
    data_queue.push(data);
    data_cond.notify_one();
  }
}

void data_processing_thread() {
  while(true) {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[]{return !data_queue.empty();});
    data_chunk data = data_queue.front();
    data_queue.pop();
    lk.unlock();
    process(data);
    if(is_last_chunk(data))
      break;
  }
}
```

Queue used to pass data

Example

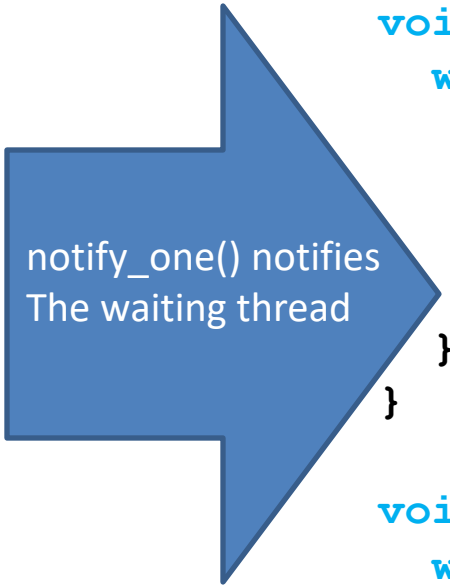
When data is ready, thread locks mutex, pushes data, and calls notify_one()

```
std::mutex mut;
std::queue<data_chunk> data_queue;
std::condition_variable data_cond;

void data_preparation_thread() {
  while( more_data_to_prepare() ) {
    data_chunk data = prepare_data();
    std::lock_guard<std::mutex> lk(mut);
    data_queue.push(data);
    data_cond.notify_one();
  }
}

void data_processing_thread() {
  while(true) {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[]{return !data_queue.empty();});
    data_chunk data = data_queue.front();
    data_queue.pop();
    lk.unlock();
    process(data);
    if(is_last_chunk(data))
      break;
  }
}
```

Example




```
std::mutex mut;
std::queue<data_chunk> data_queue;
std::condition_variable data_cond;

void data_preparation_thread() {
  while( more_data_to_prepare() ) {
    data_chunk data = prepare_data();
    std::lock_guard<std::mutex> lk(mut);
    data_queue.push(data);
    data_cond.notify_one();
  }
}

void data_processing_thread() {
  while(true) {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[]{return !data_queue.empty();});
    data_chunk data = data_queue.front();
    data_queue.pop();
    lk.unlock();
    process(data);
    if(is_last_chunk(data))
      break;
  }
}
```

Example

```
std::mutex mut;
std::queue<data_chunk> data_queue;
std::condition_variable data_cond;

void data_preparation_thread() {
  while( more_data_to_prepare() ) {
    data_chunk data = prepare_data();
    std::lock_guard<std::mutex> lk(mut);
    data_queue.push(data);
    data_cond.notify_one();
  }
}

void data_processing_thread() {
  while(true) {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[]{return !data_queue.empty();});
    data_chunk data = data_queue.front();
    data_queue.pop();
    lk.unlock();
    process(data);
    if(is_last_chunk(data))
      break;
  }
}
```

Receiver thread puts itself into waiting mode through this call (if queue is empty). It will also release the lock

Example

```
std::mutex mut;
std::queue<data_chunk> data_queue;
std::condition_variable data_cond;

void data_preparation_thread() {
  while( more_data_to_prepare() ) {
    data_chunk data = prepare_data();
    std::lock_guard<std::mutex> lk(mut);
    data_queue.push(data);
    data_cond.notify_one();
  }
}

void data_processing_thread() {
  while(true) {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[]{return !data_queue.empty();});
    data_chunk data = data_queue.front();
    data_queue.pop();
    lk.unlock();
    process(data);
    if(is_last_chunk(data))
      break;
  }
}
```

It also passes a wake condition that will be checked upon notify_all()

Example

```
std::mutex mut;
std::queue<data_chunk> data_queue;
std::condition_variable data_cond;

void data_preparation_thread() {
  while( more_data_to_prepare() ) {
    data_chunk data = prepare_data();
    std::lock_guard<std::mutex> lk(mut);
    data_queue.push(data);
    data_cond.notify_one();
  }
}

void data_processing_thread() {
  while(true) {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[]{return !data_queue.empty();});
    data_chunk data = data_queue.front();
    data_queue.pop();
    lk.unlock();
    process(data);
    if(is_last_chunk(data))
      break;
  }
}
```

The mutex will be automatically locked once the wait terminates

Example

```
std::mutex mut;
std::queue<data_chunk> data_queue;
std::condition_variable data_cond;

void data_preparation_thread() {
  while( more_data_to_prepare() ) {
    data_chunk data = prepare_data();
    std::lock_guard<std::mutex> lk(mut);
    data_queue.push(data);
    data_cond.notify_one();
  }
}

void data_processing_thread() {
  while(true) {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[]{return !data_queue.empty();});
    data_chunk data = data_queue.front();
    data_queue.pop();
    lk.unlock();
    process(data);
    if(is_last_chunk(data))
      break;
  }
}
```