COMP6771
Advanced C++ Programming

Week 9
Multithreading (continued)

2016

www.cse.unsw.edu.au/~cs6771

So Far

- Program Workflows: Sequential, Parallel, Embarrassingly Parallel
- Memory: Shared Memory, Local Memory, Message Passing
- Atomic Operations: Generally one CPU clock cycle
- Concepts and Problems: Race Conditions, Mutual Exclusion, Mutex Objects and Deadlocks

threadLamdaRace.cpp

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

// NOTE: do not compile this with -O2 it optimises out the ++
// call and prevents the race condition from happening.

int main() {
  int i = 1;
  const long numIterations = 1000000;
  std::thread t1{[&] {
    for (int j = 0; j < numIterations; ++j) {
      i++;
    }
  }};
  std::thread t2{[&] {
    for (int j = 0; j < numIterations; ++j) {
      i--;
    }
  }};
  t1.join();
  t2.join();
  std::cout << i << std::endl;
}
```

threadLamdaRace.cpp Output

```
$ ./threadLamdaRace
847046
$ ./threadLamdaRace
-18494
$ ./threadLamdaRace
25156
$ ./threadLamdaRace
-433633
$ ./threadLamdaRace
-8622
$ ./threadLamdaRace
-365615
$ ./threadLamdaRace
364766
$ ./threadLamdaRace
1
$ ./threadLamdaRace
1
$ ./threadLamdaRace
1000001
```

Atomic Operations

- Atomic types automatically synchronise and prevent race conditions
- The C++11 standard provides atomic primative types:

Typedef name	Full specialisation
atomic_bool	atomic<bool>
atomic_char	atomic<char>
atomic_int	atomic<int>
atomic_uint	atomic<unsigned int>
atomic_long	atomic<long>

Atomic Operations Example

Our problematic race condition fixed with an atomic int:

```
#include <iostream>
#include <thread>
#include <atomic>
int main() {
  std::atomic<int> i {1};
  const long numIterations = 1000000;
  std::thread t1([&i,numIterations] {
    for (int j = 0; j < numIterations; ++j) {
      i++;
    }
  });
  std::thread t2([&i,numIterations] {
    for (int j = 0; j < numIterations; ++j) {
      i--;
    }
  });
  t1.join();
  t2.join();
  std::cout << i << std::endl;
}
```

Is this a good approach?

Atomic Performance Problem

- The purpose of multi-threaded code is to improve performance
- Atomic operations slow down our code as they need to synchronise through locking
- How you structure multi-threaded code has a large influence on performance
- In our example we lock every time we need to increment, how might we do this better?

Improved Performance Example

```
int main() {
  std::atomic<int> i {1};
  const long numIterations = 1000000;
  std::thread t1([&i,numIterations] {
    int k = 0; // use a local variable
    for (int j = 0; j < numIterations; ++j) {
      k++;  // generally more complex math
    }
    i += k;
  });
  std::thread t2([&i,numIterations] {
    int k = 0;
    for (int j = 0; j < numIterations; ++j) {
      k--;
    }
    i -= k;
  });
  t1.join();
  t2.join();
  std::cout << i << std::endl;
}
```

Only write to the atomic int once per thread!

Atomic Operations Library

- The standard defines a number of operations on atomic types.
- These operations will be faster than your code.
- Example, `fetch_add()`:

```
#include <iostream>
#include <atomic>
#include <thread>

int main() {
  std::atomic<int> i{10};
  std::cout << "Initial Value: " << i << std::endl;
  int fetchedValue = i.fetch_add(5);
  std::cout << "Fetched Value: " << fetchedValue << std::endl;
  std::cout << "Current Value: " << i << std::endl;
}
```

- Ouput:

```
Initial Value: 10
Fetched Value: 10
Current Value: 15
```

Atomic Classes

- The std::atomic library does not provide much support for creating atomic wrappers around your own data types.
- It can wrap a `TriviallyCopyable type T` which is a class with no user defined copy controllers.
- It can also wrap raw pointer types (but dereferences are not atomic!).
- See: `http://en.cppreference.com/w/cpp/atomic/atomic`
- See: `http://stackoverflow.com/questions/15886308/stdatomic-with-custom-class-c-11`
- Problem: Synchonising on a whole class doesn't seem very efficent, what if only part of a class had race conditions?

C++11 Mutexes

- C++11 provides Mutex objects in the <mutex> header file.
- General idea:
  - A thread wants to read/write shared memory tries to lock the mutex object.
  - If another thread is currently locking the same mutex the first thread waits until the thread is unlocked or a timer expires.
  - When the thread obtains the lock it can safely read/write the shared memory
  - When the thread has finished using the shared memory it releases the lock
- Note:
  - If two or more threads are waiting for a lock to be released there is no order to which one will obtain the lock next.
  - It is assumed that all threads use the mutex to correctly access the shared memory.

std::mutex

- Non-timed mutex class
- Member functions:
  - `lock()` Tries to obtain the lock on the mutex and blocks indefinitely until the lock has been aquired.
  - `try_lock()` Tries to obtain the lock on the mutex, if the mutex is already locked will immediately return false, if the lock is obtained will return true.
  - `unlock()` Releases the lock currently held.
- Note: use `std::recursive_mutex` if your code is recursive and the same thread tries to repeatedly lock the same mutex.

std::mutex example

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

int main() {
  int i = 1;
  const long numIterations = 1000000;
  std::mutex iMutex;
  std::thread t1([&] {
    for (int j = 0; j < numIterations; ++j) {
      iMutex.lock();
      i++;
      iMutex.unlock();
    }
  });
  std::thread t2([&] {
    for (int j = 0; j < numIterations; ++j) {
      iMutex.lock();
      i--;
      iMutex.unlock();
    }
  });
  t1.join();
  t2.join();
  std::cout << i << std::endl;
}
```

Mutex Performance Problem

- We now have the same problem we had with atomic operations!
- `std::mutex` locking takes exclusive ownership over objects.
- Our program slows down while threads are "busy waiting" to acquire mutex locks.
- Exclusive ownership of a lock over a critical section is required for safely writing data, but what about reading it?
- Readers-writers locks allow multiple threads to read data in parallel (shared ownership) but if data needs to be written exclusive ownership over the mutex must be aquired.

Readers-writers lock (C++14)

- C++14 `std::shared_timed_mutex` is a form of Readers-writers lock
- Shared ownership (read lock): `lock_shared()` blocks until the read lock is aquired. Multiple threads can `lock_shared()` the mutex in parallel.
- Exclusive ownership (write lock): `lock()` blocks until the write lock is aquired. Both the read and write locks need to be unlocked before ownership can be aquired.
- While a thread has exclusive ownership no other thread can obtain a lock (read or write).
- Unlocking: make sure you unlock the correct lock: `unlock()` and `unlock_shared()`

Exceptions and Locking

What happen to the mutex lock if an exception is thrown out of a critical section?

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

int main() {
  int i = 1;
  const long numIterations = 1000000;
  std::mutex iMutex;
  std::thread t1([&] {
    for (int j = 0; j < numIterations; ++j) {
      iMutex.lock();
      i++;
      iMutex.unlock();
    }
  });
  std::thread t2([&] {
    for (int j = 0; j < numIterations; ++j) {
      iMutex.lock();
      if ( i == 0 ) throw ``i cannot be less than 0'';
      i--;
      iMutex.unlock();
    }
  });
  t1.join();
  t2.join();
  std::cout << i << std::endl;
}
```

Exceptions and Locking

- Major Problem: the mutex lock is never released if we throw out of a critical section.
- Stack unwinding occurs until the exception is caught. But only inside this thread. If exception isn't caught then program terminates.
- Even if the exception is caught, if the mutex isn't unlocked then deadlocks and/or undefined program behaviour may occur.
- We've seen these types of issues before with dynamic memory. How did we fix them?

Lock Guards

RAII wrapper class around a mutex.

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

int main() {
  int i = 1;
  const long numIterations = 1000000;
  std::mutex iMutex;
  std::thread t1([&] {
    for (int j = 0; j < numIterations; ++j) {
      std::lock_guard<std::mutex> guard(iMutex);
      i++;
    }
  });
  std::thread t2([&] {
    for (int j = 0; j < numIterations; ++j) {
      std::lock_guard<std::mutex> guard(iMutex);
      i--;
    }
  });
  t1.join();
  t2.join();
  std::cout << i << std::endl;
}
```

Lock Guards

- Lock guards are similar to smart pointers.
- On construction they take ownership over (lock) a mutex.
- When they go out of scope they get destroyed. Their destructor unlocks the mutex.
- Are exception safe!
- You should never manually lock/unlock a mutex - always use lock guards

Lock guard limitations

- Scenario consider a box holding x elements:

```
struct Box {
  explicit Box(int num) : num_things{num} {}

  int num_things;
  std::mutex m;
};
```

- and a function to transfer items between two boxes.

```
void transfer(Box &from, Box &to, int num) {
  std::lock_guard<std::mutex> lock1(from.m);
  std::lock_guard<std::mutex> lock2(to.m);

  from.num_things -= num;
  to.num_things += num;
}
```

- Problem what happens if two threads in parrallel attempt to move between the same two boxes in opposite directions?

Lock guard limitations

Problem: what happens if two threads in parrallel attempt to move between the same two boxes in opposite directions?

```
int main() {
  Box acc1(100);
  Box acc2(50);

  std::thread t1(transfer, std::ref(acc1), std::ref(acc2), 10);
  std::thread t2(transfer, std::ref(acc2), std::ref(acc1), 5);

  t1.join();
  t2.join();

  std::cout << acc1.num_things << std::endl;
}
```

Remember: a `std::lock_guard` locks a mutex on construction and releases on destruction.

std::unique_lock

- Possible deadlocks can occur with multiple std::lock_guard objects
- `std::unique_lock` is a more complex lock guard
- Can defer locking to avoid deadlocks

```
void transfer(Box &from, Box &to, int num) {
  // don't actually lock the mutexs yet
  std::unique_lock<std::mutex> lock1(from.m, std::defer_lock);
  std::unique_lock<std::mutex> lock2(to.m, std::defer_lock);

  // lock both unique_locks without deadlock
  std::lock(lock1, lock2);

  from.num_things -= num;
  to.num_things += num;
}
```

Example (modified) from:
http://en.cppreference.com/w/cpp/thread/unique_lock