COMP6771
Advanced C++ Programming

Week 10
Multithreading - Producer/Consumer Problem

2016

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

Recap: 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

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.

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;
}
```

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;
}
```

Producer-Consumer Problem

- Scenario: Consider a manufacturing plant which consumes the raw materials to produce products
- The manufacturing plant has limited storage space for both raw materials and final products.
- Trucks deliver raw materials to the plant, however, they arrive at random time intervals.
- If there is no space for the raw materials the trucks have to wait for space to become available.
- Trains remove the final products from the plant, if there are no products available they wait for one to be manufactured.

Producer-Consumer Problem

```
class ManufacturingPlant {
public:
  ManufacturingPlant() : materialsCount{0} {}

  void receiveMaterials(int i);
  int produceProduct();
private:
  int materialsCount;
  std::mutex materialsCountMutex;
  const int CAPACITY = 100;
};
```

Producer-Consumer Problem

```
class Truck {
public:
  Truck(ManufacturingPlant& m) : mp{m} {}
  void deliverMaterials() {
    mp.receiveMaterials(10);
  }
private:
  ManufacturingPlant& mp;
};

class Train {
public:
  Train(ManufacturingPlant& m) : mp{m} {}
  void getProduct() {
    mp.produceProduct();
  }
private:
  ManufacturingPlant& mp;
};
```

Producer-Consumer Problem

```
void ManufacturingPlant::receiveMaterials(int i) {
  std::lock_guard<std::mutex> lg(materialsCountMutex);
  if (materialsCount + i > CAPACITY) {
    // TODO: wait for capacity!
  }
  materialsCount += i;
}

int ManufacturingPlant::produceProduct() {
  std::lock_guard<std::mutex> lg(materialsCountMutex);
  if (materialsCount - 10 > 0) {
    // TODO: wait for materials to arrive
  }
  materialsCount -= 10;
  return 10;
}
```

Problem: how do we release the lock and wait for either capacity or materials?

Condition Variables

- Condition variables allow threads to block, release a mutex, and wait until data is set by another thread or until a time period has elapsed.
- Explicit inter-thread communication.
- When we need to check/wait for a condition to be true, we call `wait()` on the condition variable.
- If we need to signal (communicate) that some data has been made available we need to `notify_one()` or `notify_all()` on the condition variable.
- Need to add to the class:

```
  // used to signal the arrival of materials
  std::condition_variable hasMaterials;
  // used to signal the removal of materials
  std::condition_variable hasCapacity;
```

Producer-Consumer with Condition Variables

```
void ManufacturingPlant::receiveMaterials(int i) {
  std::unique_lock<std::mutex> lg(materialsCountMutex);
  hasCapacity.wait(lg, [this, &i] {
    if (materialsCount + i > CAPACITY) return false;
    return true;
  });
  materialsCount += i;
  hasMaterials.notify_one();
}

int ManufacturingPlant::produceProduct() {
  std::unique_lock<std::mutex> lg(materialsCountMutex);
  hasMaterials.wait(lg, [this] {
    if (materialsCount - 10 < 0) return false;
    return true;
  });
  materialsCount -= 10;
  hasCapacity.notify_one();
  return 10;
}
```

A further example is here:

http://baptiste-wicht.com/posts/2012/04/

Futures

- Problem: we've looked at lots of solutions to prevent deadlocks and memory corruption. But we're still stuck with busy waiting blocks.
- How do we make our code asynchronous and minimise waiting?
- e.g., If our train finds that it has to wait for a product to be manufactured, could it go and collect another product from a different manufacturing plant and then return to the first at a later period?

std::async

- `std::async` is used to create a thread and return a std::future which the result of the thread is stored in (e.g. it can work entirely entirely without shared memory and mutexes).
- `std::async` can hand over control of when to start a thread to the runtime system. It may not be called immediately if the system is busy.
- The `std::future` object can be used to check if the result of the `std::async` is available.

Example sketch

```cpp
#include <iostream>
#include <future>
#include <thread>
#include <chrono>

int calculate() {
  return 123;
}

int main() {
  std::future<int> fut = std::async(calculate);
  // note can force to launch a new thread using:
  // std::future<int> fut = std::async(std::launch::async,calculate);

  bool doOtherWork = true;
  while (doOtherWork) {
    // check if the result is available.
    if (fut.wait_for(std::chrono::seconds(0)) == std::future_status::timeout) {
      // do other work.. e.g. go to a different factory.
    } else {
      // either the result is available or the launch has been deferred
      doOtherWork = false;
    }
  }

  int res = fut.get(); // get the result from the future
  std::cout << res << std::endl;
}
```

Modified example from: http://en.cppreference.com/w/cpp/thread/future/wait_for

Throwing exceptions across threads

- Futures can be used to transport exceptions across threads.
- When you call `get()` on the future you may get the result or the exception thrown.

```
int calculate() {
  throw std::runtime_error("Exception thrown from thread");
}
```

```
try {
  int res = fut.get(); // get the result from the future
  std::cout << res << std::endl;
} catch (const std::exception& ex) {
  std::cout << "Exception caught" << std::endl;
}
```

Readings

- Chapter 23 Professional C++
- http://baptiste-wicht.com/posts/2012/03/cp11-concurrency-tutorial-part-2-protect-shared-data.html