Multithreading is the answer. What is the question?

(part 1)

Ansel Sermersheim

CppNow - May 2016

Introduction

- What is multithreading
- Terminology
- Problems multithreading solves
- When is multithreading the answer?
- How to review multithreaded code

Introduction

- I have read the C++ standard, now what?
- Which flour do you use in a Toll House cookie?
  - bread flour, cake flour, all purpose flour
- Just slap a lock on it and call it done

What is Multithreading

- Bob is a seasoned C++ programmer
- Bob has a complex issue to solve
- Bob has always wanted to use multithreading
- Unfortunately, Bob now has N issues to solve
  - at least one race condition
  - a few memory leaks
  - and a random runtime crash which will only be found by a high profile customer

What is Multithreading

- If you want to learn multithreading, find a problem which actually requires a multithreaded solution

- Do not take your current problem and force multithreading to be the solution

What is Multithreading

- Multithreading is the ability of a program to execute multiple instructions at the same time
  - a mechanism by which a single set of code can be used by several threads at different stages of execution
  - the ability to execute different parts of a program simultaneously
  - Multithreading may be considered as concurrency if the threads interact or parallelism if they do not

What is Multitasking

- Multitasking is the concept of performing multiple tasks or processes over a certain period of time by executing them concurrently
  - does not automatically imply multithreading
  - on a single processor system multitasking is implemented by time slicing and the CPU switches between different tasks

Multithreading - Myths

- Threading in a multiprocessor system results in concurrent execution and makes a program faster
- Multithreading improves the stability of programs
- Since each thread is handled separately, if one thread has an error, it does not affect the rest of the program
- A truly smart programmer will not have issues writing a clean multithreading program
- Multithreading is hard
- Multithreading is easy

Terminology

- Thread
  - work which can be scheduled to execute on one core
  - a thread is contained inside a process
  - each thread has its own call stack

- Process
  - used to start a separate program
  - if there is only one thread in a process the program is not multithreaded
  - as an example, you start Make then Make starts Clang, Make starts a new process to run Clang
  - threads in the same process share most resources

Terminology

- Core
  - "core count" is the total number of instructions that can be executed simultaneously
  - a computer may have multiple processors, each of which might have multiple cores
  - a thread consumes an entire core while it is active
  - more cores does not mean your program will run faster
  - not all cores are equal
    - HyperThreading, NUMA, AMP

Terminology

- Cores in a practical system
  - we upgraded our CI machine to a CPU with 6 cores
- Why not buy a CPU with more cores
  - the next higher model of CPU has 8 cores
  - however the 8 core CPU has a slower clock speed
  - overall performance on the 8 core CPU is not as much of an increase as expected and not worth the cost

Terminology

- Resource
  - computer memory location
  - file handle
  - non thread-safe C++ objects

- A resource must not be accessed by multiple threads simultaneously

Terminology

- Race condition
  - occurs when a resource is accessed by multiple threads simultaneously, and at least one access is a write
  - undefined behavior

Terminology

- Stack
  - is an area of memory used for data whose size is determined at compile time
  - belongs to a specific thread
- Heap
  - an area of memory used for data which is allocated at runtime
  - shared among all threads

Terminology

- Fibers
  - a “lightweight” thread
  - fibers are not scheduled by the OS, you have to make a call to explicitly start and stop a fiber
  - current execution path is only interrupted when the fiber yields execution
  - no two fibers can run at exactly the same time
  - can be difficult to use correctly since the OS is not in charge of scheduling
- Not currently in C++
- Available in Boost.Fiber

Terminology

- Green Thread
  - a thread that is scheduled by a runtime library instead of natively by the underlying operating system
  - used to emulate multithreading without OS support
- Not currently in C++
  - most operating systems support native threads
- Not widely used outside of
  - Java (older versions)
  - Erlang (sort of)
  - Go

Multithreading solves . . .

When to use Multithreading

- Problems for which multithreading is the answer
  - tasks which can intuitively be split into independent processing steps
  - a problem where each step has a clear input and output
  - intensive computations
  - continuous access to a large read-only data set
  - processing a stream of large data files

When to use Multithreading

- Problems for which multithreading is the only answer
  - tasks whose performance would be unacceptable as a single thread
  - processes where the workload cannot be anticipated
  - manage concurrent access to multiple resources, such as an operating system
  - external clients sending requests to a process in a random and unpredictable fashion, such as PostgreSQL

When to use Multithreading

- Real life example
  - project: streaming video server
  - performance: the prototype was terrible
  - goal for production code: as fast as possible
    - assumed multithreading was the right path
    - performed a minor benchmark
    - the bottleneck turned out to be in the hardware
    - optimization was the correct solution not multithreading

Matching Problems with Solutions

- What kind of ice cream maker do I need
  - for a small dinner party
  - for an ice cream shop

- Multithreading techniques in this talk apply to:
  - C++11 or later
  - two to twenty cores
  - desktop systems, mobile devices, cloudy things

Multithreading Analogy (A)

- Stage: A kitchen
  - two chefs
    - each chef will represent a thread
  - two knives
    - each knife is a local resource

- Requirement: make 50 fruit salads
- Solution: each chef will make 25 fruit salads

Threading Code (A)

```
std::thread chef1(
  [](){
    for(int i = 0; i < 25; ++i) {
      makeFruitSalad();
    }
  }
);

// same code as for chef one
std::thread chef2(...);

chef1.join();
chef2.join();
```

Multithreading Analogy (B)

- Stage: A kitchen
  - two chefs
    - each chef will represent a thread
  - two knives
    - each knife is a local resource
  - one oven
    - shared resource

- Requirement: make 50 apple pies
- Solution: each chef will independently make 25 apple pies

Threading Code (B)

```
Oven vikingOven;
std::mutex oven_mutex;

std::thread chef1( [&oven_mutex, &vikingOven]()
  {
    for(int i = 0; i < 25; ++i) {
      Pie anotherPie;
      anotherPie.makeCrust();
      anotherPie.putApplesInPie();
      std::lock_guard<std::mutex> oven_lock(oven_mutex);
      vikingOven.bakePie(anotherPie, 375, 35);
    }
  }
);

std::thread chef2(...);

chef1.join();
chef2.join();
```

Multithreading Analogy (C)

- Stage: A kitchen
  - two chefs
    - each chef will represent a thread
  - two knives
    - each knife is a local resource
  - one oven
    - shared resource

- Requirement: make 50 apple pies
- Solution: one chef prepares pies, the second chef bakes the pies in the oven

Threading Code (C-1)

```
Oven vikingOven;
threadsafe_queue<Pie> conveyorBelt;

std::thread chef1( [&conveyorBelt]()
  {
    for(int i = 0; i < 50; ++i) {
      Pie anotherPie;
      anotherPie.makeCrust();
      anotherPie.putApplesInPie();

      // give the pie away
      conveyor_belt.queue(std::move(anotherPie));
    }
  }
);
```

Threading Code (C-2)

```
std::thread chef2( [&conveyorBelt, &vikingOven]()
  {
    for(int i = 0; i < 50; ++i) {
      Pie anotherPie = conveyorBelt.dequeue();

      // bakePie method is blocking
      vikingOven.bakePie(anotherPie, 375, 35);
    }
  }
);

chef1.join();
chef2.join();
```

Threading Code (C)

- Can this design be optimized?
- Can these threads cause a deadlock?
- Are there any race conditions?

Multithreading Analogy (D)

- Stage: A kitchen

- Requirement: need 25 fruit salads and 25 chicken salads
- Solutions:
  - each chef independently makes a fruit salad, cleans up, and then makes a chicken salad, 25 times
  - one chef makes only the 25 fruit salads while the other chef makes only the 25 chicken salads
  - both chefs each make the 25 fruit salads tracking how many were made in a shared data location
    - as soon as the fruit salads are finished they both switch to making chicken salads

Multithreading Analogy (E)

- Stage: A kitchen
  - one oven, one brick pizza oven, one ice cream maker
    - shared resources

- Requirement:
  - anyone can randomly order pizza, garlic knots, apple pie, or ice cream

- Solution: pandemonium

Multithreading Analogy (E)

```
Oven vikingOven;
std::mutex vikingOven_mutex;

Oven brickOven;
std::mutex brickOven_mutex;

IceCreamMaker iceCreamMaker;
std::mutex iceCream_maker_mutex;

class Food  { ... };

class Pizza { ... };
class GarlicKnots { ... };
class ApplePie { ... };
class IceCream { ... };
```

Multithreading Analogy (E)

```
void eat(Food && food) {
  std::cout << "Patron was served: " << food.name();
};

using PatronTicket = std::future<std::unique_ptr<Food>>;
using ChefTicket   = std::promise<std::unique_ptr<Food>>;
```

Multithreading Analogy (E)

```
std::thread patron1( []() {
    PatronTicket knots    = orderGarlicKnots();
    PatronTicket pizza    = orderPizza();
    PatronTicket iceCream = orderIceCream();

    eat(knots.get());
    eat(pizza.get());
    eat(icecream.get());
});

std::thread patron2( []() {
    PatronTicket iceCream = orderIceCream();
    PatronTicket applePie = orderApplePie();

    eat(iceCream.get());
    eat(applePie.get());
});
```

Multithreading Analogy (E)

```
class Order { ... };
std::atomic<bool> restaurantOpen;
threadsafe_queue<Order> orderQueue;

std::thread chef1( [&]() {
  while(restaurantOpen) {
    Order nextOrder = orderQueue.dequeue();
    nextOrder.process();
  }
});

std::thread chef2( [&]() {
  while(restaurantOpen) {
    Order nextOrder = orderQueue.dequeue();
    nextOrder.process();
  }
});
```

Multithreading Analogy (E)

```
PatronTicket orderPizza() {
  std::shared_pointer<ChefTicket> chefTicket =
          std::make_shared<ChefTicket>();
  PatronTicket patronTicket = chefTicket->get_future();

  Order order{ [chefTicket]() {
    std::unique_ptr<Pizza> pizza = std::make_unique<Pizza>();
    pizza->addSauce();
    pizza->addCheese();
    std::lock_guard<std::mutex> lock(brickOven_mutex);
    pizza = brickOven.bake(std::move(pizza));
    chefTicket->set_value(std::move(pizza));
  }};

  orderQueue.queue(std::move(order));
  return patronTicket;
}
```

Multithreading Analogy (E) C++14

```
// changes to the lambda to move capture

PatronTicket orderPizza() {
  ChefTicket chefTicket;
  PatronTicket patronTicket = chefTicket->get_future();

  Order order{ [captureTicket = std::move(chefTicket)] () {

     . . .

  }
```

Multithreading Analogy (E)

- Items to consider about this example
  - single queue is not efficient
    - one queue per thread will improve performance
    - an idle thread can steal work from other queues, this is called “work stealing” and is a common feature
  - a chef should not be waiting for a pizza to bake
  - locking should not be arbitrary
    - `std::lock_guard<std::mutex> lock(brickOven_mutex);`
    - there should be a better way ...

Miscellaneous threading advice

- Too many active threads
  - one active thread per core is ideal
  - move blocking calls to extra threads which can then wait, without stalling the rest of the program

- Too much shared data
  - concentrate your efforts on reducing the number of shared data structures
  - reduce the size of each shared data structure
  - reduction of shared data should drive the entire design
  - read-only shared data is much better than writable shared data

Miscellaneous threading advice

- A race condition implies a write to shared data
  - no shared data means no race conditions
  - read-only shared data means no race conditions

Multithreading: There is a better way

- Please stay for my next talk...

Wrap Up

Libraries & Applications

- CopperSpice
  - libraries for developing GUI applications
- PepperMill
  - converts Qt headers to CS standard C++ header files
- CsSignal Library
  - new standalone thread aware signal / slot library
- LibGuarded
  - new standalone multithreading library for shared data

Libraries & Applications

- KitchenSink
  - one program which contains 30 demos
  - links with almost every CopperSpice library

- Diamond
  - programmers editor which uses the CS libraries

- DoxyPress & DoxyPressApp
  - application for generating documentation

Where to find these Projects

- www.copperspice.com
- download.copperspice.com
- forum.copperspice.com

- ansel@copperspice.com

- Questions? Comments?