Concurrent Thinking

Anthony Williams

Just Software Solutions Ltd
https://www.justsoftwaresolutions.co.uk

23rd April 2016

Concurrent Thinking

- What is Concurrent Thinking?
- Race conditions
- Synchronization tools
- Designing for concurrency — Concurrent Thinking in practice

What is Concurrent Thinking?

What is Concurrent Thinking?

Concurrent Thinking

The mindset and thought processes used for analysing and designing systems with multiple concurrent execution streams.

Essential Aspects of Concurrent Thinking

- Which processing can be done independently?

Essential Aspects of Concurrent Thinking

- Which processing can be done independently?
- What are the interactions between execution streams?

Essential Aspects of Concurrent Thinking

- Which processing can be done independently?
- What are the interactions between execution streams?
- Is access to shared state correctly synchronized?

Essential Aspects of Concurrent Thinking

- Which processing can be done independently?
- What are the interactions between execution streams?
- Is access to shared state correctly synchronized?
- Is there a potential for race conditions?

Race Conditions

Race Condition

A situation in which the behaviour of a system with concurrent execution streams depends on the relative speeds of processing or the scheduling of those execution streams.

Race conditions may be **problematic** or **benign**.

Benign Race Condition

```
void func(int num){
  char buffer[20];
  sprintf(buffer,"thread %i\n",num);
  std::cout<<buffer;
}

int main(){
  std::thread t1(func,1);
  std::thread t2(func,2);
  t1.join(); t2.join();
}
```

Benign Race Condition

The output is:

```
thread 1
thread 2
```

or

```
thread 2
thread 1
```

Either is OK, so this is benign.

Data Race

A **problematic race condition** where a **write** to some non-atomic shared data from one execution stream races with **any access** to that shared data from another execution stream **without synchronization**.

Data races are **always undefined behaviour**.

Data Races

```
unsigned i=0;
void func()
{
    for(unsigned c=0;c<2000000;++c)
        ++i;
    for(unsigned c=0;c<2000000;++c)
        --i;
}
int main(){
  std::thread t1(func),t2(func);
  t1.join(); t2.join();
  std::cout<<"Final i="<<i<<std::endl;
}
```

Data Races

```
Final i=0
```

Data Races

```
Final i=0

Final i=4294345393

Final i=169708
```

Undefined Behaviour

Synchronization Tools

Synchronization Tools

In C++, we have several tools at our disposal to synchronize data:

- Mutexes and condition variables
- Futures
- Latches and Barriers
- Atomic types
- Third party tools — thread pools, actor libraries, concurrent data structures etc.

Mutexes

Mutex: **Mut**ual **Ex**clusion

A mutex is a means of **preventing** concurrent execution.

Mutexes

Thread 1	**Thread 2**
Lock mutex m *succeeds*	Lock mutex m *blocks*
Do stuff	
Unlock mutex m	
	unblocked *lock operation returns*
	Do stuff
	Unlock mutex m

Condition Variables

`std::condition_variable` is a **notification mechanism** to avoid busy waits.

Condition Variables

`std::condition_variable` is a **notification mechanism** to avoid busy waits.

Spurious wakes are a BIG source of bugs.

Condition Variables

`std::condition_variable` is a **notification mechanism** to avoid busy waits.

Spurious wakes are a BIG source of bugs.

Missed notifications are also a BIG source of bugs.

Condition Variables

`std::condition_variable` is a **notification mechanism** to avoid busy waits.

Spurious wakes are a BIG source of bugs.

Missed notifications are also a BIG source of bugs.

You need to write your code to work if none of the functions do anything, **and** if the notifications only wake the minimum threads they have to.

Spurious Wakes

```
std::mutex m;
std::condition_variable cv;
int data;

void process(int);
void foo(){
  std::unique_lock<std::mutex> lk(m);
  cv.wait(lk);
  process(data);
}
```

Spurious Wakes

```
std::mutex m;
std::condition_variable cv;
int data;

void process(int);
void foo(){
  std::unique_lock<std::mutex> lk(m);
  lk.unlock(); lk.lock();
  process(data);
}
```

Missed Wakes

```
std::atomic<bool> done(false);
void foo(){
  std::unique_lock<std::mutex> lk(m);
  cv.wait(lk,[]{return done.load();});
  process(data);
}
void signal_ready(){
  done=true;  cv.notify_one();
}
```

Missing Wakes

Thread 1	**Thread 2**
Calls `signal_ready()`	Calls `foo()`
	Locks `m`
	Reads `done` *returns* `false`
`done=true`	*Suspended by scheduler*
`cv.notify_one()`	
	Woken by scheduler
	Blocks on `cv` *Wakeup missed*

Missed Wakes

```
void foo(){
  std::unique_lock<std::mutex> lk(m);
  cv.wait(lk,[]{return done.load();});
  process(data);
}
void signal_ready(){
  done=true;
  m.lock(); m.unlock();
  cv.notify_one();
}
```

Missing Wakes

Thread 1	**Thread 2**
Calls `signal_ready()`	Calls `foo()`
	Locks `m`
	Reads `done` *returns* *`false`*
`done=true`	*Suspended by scheduler*
`m.lock()` *blocks waiting for thread 2*	
	Woken by scheduler *Blocks on* *`cv`* *and unlocks* *`m`*
unblocked `cv.notify_one()`	
	Wakeup seen

Futures

Futures provide a one-shot synchronization mechanism.

One thread provides the data via `std::promise`, `std::packaged_task`, or `std::async`.

Other threads get the data via `std::future` or `std::shared_future`.

Futures

```
std::promise<MyData> prom;
std::future<MyData> f=prom.get_future();
```

Thread 1	**Thread 2**
`do_stuff(fut.get())` *blocks*	
	`prom.set_value(foo())`
unblocked *`get()` returns* *`do_stuff()` called*	

Futures in the Concurrency TS

The Concurrency TS extends C++11 futures in two ways:

- Continuations — you may specify a task to run when the future becomes *ready* with `fut.then()`
- Waiting for collections — you can wait for one of a set of futures to become *ready* with `when_any`, or all of them to become *ready* with `when_all`

Continuations and `std::future`

- A continuation is a new task to run when a future becomes ready
- Continuations are added with the new `then` member function
- Continuation functions must take a `std::future` as the only parameter
- The source future is no longer `valid()`
- Only one continuation can be added

Continuations and `std::future`

```
int find_the_answer();
std::string process_result(
  std::future<int>);
auto f=std::async(find_the_answer);
auto f2=f.then(process_result);
```

`when_any`

`when_any` is ideal for:

- Waiting for speculative tasks
- Waiting for first results before doing further processing

```
auto f1=std::async(foo);
auto f2=std::async(bar);
auto f3=when_any(
  std::move(f1),std::move(f2));
f3.then(baz);
```

when_all

`when_all` is ideal for waiting for all subtasks before continuing. Better than calling `wait()` on each in turn:

```
auto f1=std::async(subtask1);
auto f2=std::async(subtask2);
auto f3=std::async(subtask3);
auto results=when_all(
  std::move(f1),std::move(f2),
  std::move(f3)).get();
```

Concurrency TS latches and barriers

- `latch` is a single-use count-down: once the count reaches zero it is permanently signalled.
- `barrier` and `flex_barrier` are reusable count-downs: once the count reaches zero, the barrier is signalled and the count reset.

Both latches and barriers allow threads to wait until the latch or barrier is signalled.

Atomic types

- `std::atomic<T>` provides an atomic type that can store objects of type `T`.
  - `T` can be a built in type, or a class type of any size
  - `T` must be *trivially copyable*
  - `compare_exchange_`*`xxx`* operations require that you can compare `T` objects with `memcmp`
  - `std::atomic<T>` may not be lock free — especially for large types
- `std::atomic_flag` provides a guaranteed-lock-free flag type.
- The Concurrency TS provides `atomic_shared_ptr` and `atomic_weak_ptr`.

Atomic Operations

General ops

`load()`, `store()`, `exchange()`, `compare_exchange_weak()`, `compare_exchange_strong()`

`=`

Arithmetic ops for `atomic<`*`Integral`*`>` and `atomic<T*>`

`fetch_add()`, `fetch_sub()`

`++`, `--`, `+=`, `-=`

Bitwise ops for `atomic<`*`Integral`*`>`

`fetch_and()`, `fetch_or()`, `fetch_xor()`

`&=`, `|=`, `^=`

Flag ops for `atomic_flag`

`test_and_set()`, `clear()`

Atomic Operations

General ops

`load(), store(), exchange(),`
`compare_exchange_weak(),`
`compare_exchange_strong()`
`=`

Arithmetic ops for `atomic<Integral>` and `atomic<T*>`

`fetch_add(), fetch_sub()`
`++, --, +=, -=`

Bitwise ops for `atomic<Integral>`

`fetch_and(), fetch_or(), fetch_xor()`
`&=, |=, ^=`

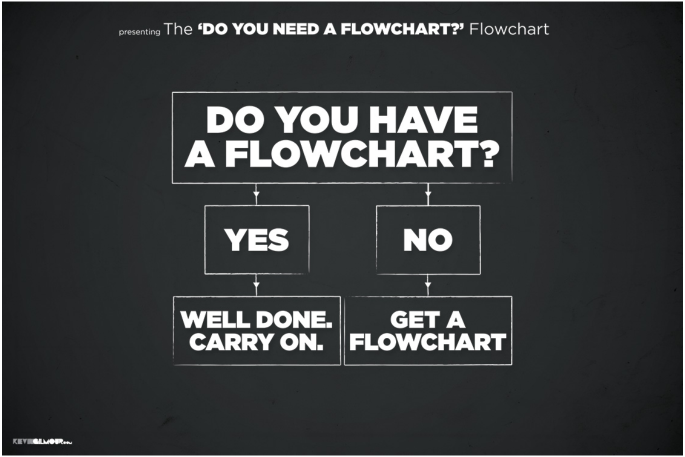
MAY NOT BE LOCK FREE

Flag ops for `atomic_flag`

`test_and_set(), clear()`

GUARANTEED LOCK FREE

Designing for Concurrency



Designing for Concurrency: Key Questions

- Which processing can be done independently?
- Which threads might be touching the same data?
- At what level should the accesses be synchronized?

Independent Processing

Independent processing means:

- No synchronization
- No problematic race conditions
- Simpler code

We want as much independent processing as possible.

Which threads might be touching the same data?

This is applicable at **all** levels of an application.

The granularity of the data varies with the code granularity.

At what level should the accesses be synchronized?

This is a question of size.

Large chunks $\Rightarrow$ more independent processing, less synchronization. Larger latencies.

small chunks $\Rightarrow$ less independent processing, more synchronization. Smaller latencies.

Analysing concurrent accesses

To check for correctness we need to analyse concurrent accesses by separate threads.

The fewer such threads there are, the easier this is.

Analysing concurrent accesses

We need to think about:

- Which data each thread accesses
- What values are read
- What is guaranteed (or not) to be visible to each thread
- All the possible places a thread can be suspended
- What reorderings the compiler or processor might do

Analysing concurrent accesses

I use sequence tables with one column per thread:

Thread 1	**Thread 2**
Lock mutex a	Lock mutex b
Lock mutex b	Lock mutex a
Blocked waiting for thread 2	*Blocked waiting for thread 1*

Analysing a queue

Consider a queue with the same interface as `std::queue`:

```
class concurrent_queue{
public:
  void push(DataType v);
  bool empty() const;
  DataType front();
  void pop();
};
```

Analysing a queue

If more than one thread can call `pop()`, this queue is broken **at the interface**.

It doesn't matter what synchronization is used internally.

Analysing a queue

Consider multiple threads calling `foo`:

```
concurrent_queue q;
void foo(){
  if(!q.empty()){
    auto data=q.front();
    q.pop();
    process(data);
  }
}
```

Analysing a queue

Thread 1	**Thread 2**
`if(!q.empty())` *returns* `false`	`if(!q.empty())` *returns* `false`
`auto data=q.front()` *The same data element is retrieved in both threads*	`auto data=q.front()`
`q.pop()` *Two elements are popped*	`q.pop()`

Avoiding racy interfaces

The solution is to adjust the interface, and group the operations that must be indivisible.

```
bool try_pop(DataType& popped_value);
```

or

```
DataType blocking_pop();
```

Analysing thread interactions

Sequence tables are easily extensibly to N threads.

Can incorporate out-of-order execution and delayed visibility $\Rightarrow$ annotate with loaded values.

Can leave a column with blank entries to indicate “not running” states, where the thread is suspended by the scheduler.

Analysing thread interactions: progress rates

A key thing to remember: each sequence table is **only one possible execution**.

Each thread may progress faster or slower than you expect

$\Rightarrow$ Add or remove empty rows in each column and consider the consequences.

Analysing thread interactions: data visibility

If one thread loads a value written by another thread, consider what other options there are.

Write a sequence table for each possibility.

Racy references

```
future<MyData> foo(ParamType p){
  auto f=async([&]{
    return get_part1(p);
  });
  auto part2=get_part2(p);
  return f.then([&](auto part1){
    return combine(
      part1.get(),part2);
  });
}
```

Racy references

```
future<MyData> foo(ParamType p){
  auto f=async([&]{
    return get_part1(p);
  });
  auto part2=get_part2(p);
  return f.then([&](auto part1){
    return combine(
      part1.get(),part2);
  });
}
```

Racy references

Dangling pointers and references are easier to get in concurrent code.

Racy references

Dangling pointers and references are easier to get in concurrent code.

They may not show up in testing due to scheduling.

Racy references

Dangling pointers and references are easier to get in concurrent code.

They may not show up in testing due to scheduling.

Pay particular attention to object lifetimes with concurrent code.

Guidelines

Essential Aspects of Concurrent Thinking

- Which processing can be done independently?
- What are the interactions between execution streams?
- Is access to shared state correctly synchronized?
- Is there a potential for race conditions?

Guidelines

- Keep threads independent where possible
- Where interactions are necessary, specify the interactions carefully
- Take full advantage of the synchronization facilities available
- Use sequence tables to analyse the behaviour of concurrent threads and avoid problematic race conditions

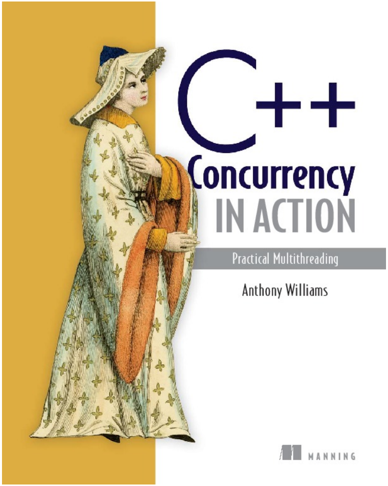
Questions?

Just::Thread

`just::thread` provides a complete implementation of the C++14 thread library and the C++ Concurrency TS.

`Just::Thread` Pro gives you actors, concurrent hash maps, concurrent queues and synchronized values.

My Book



C++ Concurrency in Action:
Practical Multithreading

`http://stdthread.com/book`

Picture credits

The images listed below are from the specified source, with the specified license. All other images are copyright Just Software Solutions Ltd, licensed under Creative Commons Attribution ShareAlike 4

- Army 10 Miler: https://flic.kr/p/8MPbEW, U.S. Army photo by Tim Hipps — Creative Commons Attribution 2
- Burnt Server: http://commons.wikimedia.org/wiki/File:Backup_Backup_Backup_-_And_Test_Restores.jpg by John — Creative Commons Attribution 2
- Brain Neurons: https://flic.kr/p/6PKTHD, Fotis Bobolas — Creative Commons Attribution ShareAlike 2
- Synchronized Clocks: https://flic.kr/p/bY1SU5 by Pete — Public Domain
- Success sign: https://commons.wikimedia.org/wiki/File:Success_sign.jpg by Keith Ramsey (RambergMediaImages), Creative Commons Attribution ShareAlike 2
- Flowchart: https://flic.kr/p/8nXUr2 by Kevin Gilmour, Creative Commons Attribution 2

Creative Commons Attribution ShareAlike 2: https://creativecommons.org/licenses/by-sa/2.0/
Creative Commons Attribution ShareAlike 4: https://creativecommons.org/licenses/by-sa/4.0/
Creative Commons Attribution 2: https://creativecommons.org/licenses/by/2.0/