Almost Functional Programming in C++

Writing scalable servers by utilizing immutable data structures

Introduction

- Record!!!
- Introduction
- This talk is the conclusion of _my_ experiences.
- Who programs C++, Multithreaded C++, C#?
- This talk is a watered down version of my APF C# article. (http://www.codeproject.com/Articles/557658/AFP-Almost-Functional-Programming-in-Csharp-Part)
- Some things are missing I wanted to talk about, very little boost::range.
- Sample project probably has some bad const and copy parameters. (https://github.com/NoiseEHC/afp-cpp-server)

Presentation

This presentation is about scaling both in performance and both in developer time (50K vs 100+K)

1. What is wrong with the “normal” way of writing MT code.
2. Going to readonly configuration.
3. Working out the details.
4. What is next.

What this talk is not

1. Not about scaling to clusters
2. Not about embarrassingly parallel problems (OpenMP, AMP)
3. Not about web servers
4. Not a magic wand
5. Not applicable to all problems*

“Normal” way of writing MT code

Normally multithreaded programs happen in these steps:

1. Run things on a single thread.
2. When we run out of MHz, run the same thing on multiple threads.
3. When we notice a race condition, put some locks around.
4. When we notice that there are locks inside locks, create a lock hierarchy.
5. When it will not scale because of all those locks, replace some with a read-write lock.
6. ???
7. Profit!

Things to avoid 1

```
{
    lock_guard<mutex> lock(_mutex);
    ... do something
}
```

When we realize that “something” takes too long time:

```
{
    lock_guard<mutex> lock(_mutex);
    ... start long running operation
}
```

... do something long running

```
{
    lock_guard<mutex> lock(_mutex);
    ... finish long running operation
}
```

Things to avoid 2

And it works until we have to do some defensive programming:

```
{
    lock_guard<mutex> lock(_mutex);
    auto iterator = _map.find(id);
    if(iterator != _map.end())
        ...
}
```

... do something long running

```
{
    lock_guard<mutex> lock(_mutex);
    _map[id].finished = true; // oops, what if it does not exist?
}
```

Things to avoid 3

This is well understood:

```
{
    lock_guard<mutex> lock(_mutex);
    _list.push_back(something);
}
```

What about this?

```
{
    lock_guard<mutex> lock(_mutex);
    for(auto const &item : _list)
        ... do something not trivial
}
```

Things to avoid 3a

Usually we turn it into this:

```
vector<...> copyoflist;
{
     lock_guard<mutex> lock(_mutex);
     copyoflist = _list;
}
for(auto const &item : copyoflist)
     ... do something not trivial
```

Things to avoid 3b

Usually we hope in read-write locks:

```
{
    shared_lock<shared_timed_mutex> lock(_sharedMutex);
    for(auto const &item : _list)
        ... do something not trivial
}
```

Unfortunately pending write locks block reads… Also slow.

Things to avoid 4

- Lock hierarchies. Lock state is a leaky abstraction!
- Separate processing steps with arbitrarily chosen number of threads processing them.
- Server slices…
- Writing configuration UI after the server is finished.

Going to readonly “configuration”

- “Configuration” means anything whose mutation is not the purpose of our program, but infrequently changes. Like number of frames, chain length or current batch. Like server list. Like temp folder.
- The program works by taking the current state (of the configuration), then creates a new, modified configuration, then replaces the old state with the new state. Repeat.
- For simplicity we just recreate everything from scratch, but you can share identical parts easily.
- In Clojure it is handled by persistent data structures. Unfortunately they are not implemented in C++. Yet.

The toy problem: trading

- Calculate the value of our portfolios (bunch of stock, price depends on currency exchange too).
- Everything which is sequential has to be executed in one go, there is no point passing calculation from thread to thread unless we fan out.
- Stock market is simulated by random prices, some says that it is a good enough model of reality.
- We just print the price to the console, but normally we should send it to the trading server or the exchange.

The toy problem: “configuration”

- There are a list of portfolios.
- Each portfolio has a list of stocks (stockId, number).
- There are a list of stocks (or currencies, currencyId).
- The configuration has pointers which may be wrong.
- Configuration UI is simulated by reloading the xml file.

Look at code, and sample config.xml.

The toy problem: processing

- We receive a new price (but not from the network in the sample).
- Process everything synchronously what can be done.
- Post all fanout processing to boost::asio::io_service::strand. (It is a good question whether it keeps the order…)
- Process all posted processing synchronously unless another fanout or fanin.

Look at FakeMarketData, Portfolio.

The toy problem: configuration changes

- If any of the portfolios reference a stock, or the stock references a currency, then the stock or currency has to be Subscribed.
- If the stock or currency are not referenced anymore, it has to be Unsubscribed.
- If there is a new portfolio, it has to be created and made depend on the stocks (and their currencies).
- If the portfolio is deleted, the dependency has to be removed.
- Handle if the portfolio's list of stocks changes. (We actually forget everything as this talk does not include AFP Part 1, but maybe next time...)
- **Start, Change and Shutdown are exactly the same thing!!!**

Working out the details

Querying and updating the configuration is easy in C#, but hard in C++.

```
static volatile GlobalState _currentState;
...
GlobalState state = _currentState;
...
_currentState = newState;
```

In C++ this is much harder, so simple solution for now:

```
std::shared_ptr<GlobalState> _currentState;

auto state = atomic_load_explicit(&_currentState, memory_order_acquire);

atomic_store_explicit(&_currentState, newState, memory_order_release);
```

Working out the details, continued

- Everything is either synchronously executed or posted to boost::asio::io_service::strand.
- Possible optimization in packet queuing - in fanout post everything except one.
- There are a few rare places when it might make sense of pipelined execution.
- I/O without locking requires posting next read after order dependent execution finished.

Categorize

We have to compare the old and new configuration. Compare apples to oranges!
Left is the freshly loaded configuration.
Right is the old one which is the currently running stateful objects.
They both have an std::string Id;

P1 (VOD LN)	Left only.	No matching Id.	New portfolio, create it.
P2 (VOD LN)	Both, same.	P2 (VOD LN)	Do nothing.
P3 (BP LN)	Both, different.	P3 (VOD LN)	Change stocks, delete old, create new.
	Right only.	P4 (BMW GY)	Deleted, delete it.

Look at Categorize<>, we call it over and over and over again.

PerformConfigChange

1. Calculate new state in ConfigChange(). Anything what fails before the PerformConfigChange() call, does not modify the current state. Everythig is exception safe.
2. In PerformConfigChange stop (Unsubscibe) things before deleting the objects those running packets use. Before atomic_store_explicit(&_currentState…), anything fails will make the server partially stopped but does not crash it.
3. Exchange _currentState with the new state. Immediately all new packets will use the new global state, but the aready running ones use the old one.
4. Start the new objects (Subscribe), those will create running packets which reference the new state. If anything fails here, the failed state shoul be stored in the object to retry later (this is not in the sample).

What is next

- Another ½ hour with boost::range and “finished” sample.
- Real atomic_load_explicit(&_currentState, memory_order_acquire) implementation.
- Real ThreadPool & ThreadPoolQueue implementation.

Further reading

CPPCON 2015: How to make your data structures wait-free for reads
https://www.youtube.com/watch?v=FtaD0maxwec
http://concurrencyfreaks.blogspot.co.nz/2013/12/left-right-classical-algorithm.html

CPPCON 2015: Parallel Program Execution using Work Stealing
https://www.youtube.com/watch?v=iLHNF7SgVN4

Using Quiescent States to Reclaim Memory
http://preshing.com/20160726/using-quiescent-states-to-reclaim-memory/

Questions?