Concurrency in C++, Part I

June 1st, 2019

Overview

- Course 1: C++ Foundations
  - Ex algorithm: A* (and thus priority queues)
- Course 2: Object-Oriented Programming
  - Ex application: system monitor...
- Course 3: Memory Management
  - SMART pointers
- Course 4: this->course = concurrency
  - Part I: introduction
    - Why concurrency?
    - What will you achieve?
    - 3rd week preview: First parallel algorithm...
    - Your project: Chatbot
- Capstone work - final nanodegree project

Why concurrency?

Speed

- Faster A*
- Faster algorithms
- Use faster algorithms
- (Make Machine Learning and AI possible, BTW)
- **How to check speed up or latency improvement is there...**

Integration

- Even if not writing parallel algorithms per se, your target platform might require it
- How to make algorithms work together
- **How to (try to) test & debug**

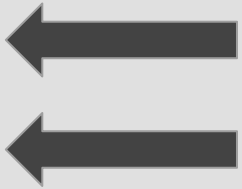
Everywhere

(Ex: ESP32 DevKitC ver. D)

- 2x Xtensa LX6 Cores
- 520 KB RAM
- 4MB Flash

Many ways to parallelize...

- Language support: OpenMP/Cuda/OpenACC
- Task/thread pool based
- Lock synchronization
- Lock-free
- STM/HTM
- Message passing
- Others...



What will you achieve?

Goals

- Master the use of parallel algorithms from standard libraries
- Understand how to test and how to analyse a new parallel algorithm
  - Why this is not easy
  - Why you should either avoid it or do it very seriously
  - Some tools & methods to be serious
- Write a simple enough but also fast enough application: concurrent chat engine

Threads & thread synchronization cursus (preview)

Week two: threads

- std::thread
- std::async/std::future
- Some design patterns
- Some debugging tips
- More design patterns (to help debugging)
- Measuring time (again)
- Logging
- Technicalities: CPU affinity, error management (if possible at all), ...

Week three: thread synchronization

- Locks & condition variables
- Some design patterns
- Introduction to lock-free types & algorithms
- **More design patterns**
- **More debugging**
- **Some ways to analyse algorithms beforehands (and thus save debugging time)**

This week: standard algorithms, in the standard library

From this:

```
std::vector<double> v(10'000'007, 0.5);

double result = std::accumulate(v.begin(), v.end(), 0.0);
```

To this:

```
std::vector<double> v(10'000'007, 0.5);

double result = std::reduce(std::execution::par, v.begin(), v.end());
```

This week: standard algorithms, in the standard library

- We will want to write as much as possible of our code as this pattern

```
auto result = ALGORITHM(EXECUTION_POLICY, FROM, TO, ARGS...);
```

- This will be parallel when EXECUTION_POLICY is, typically: ***std::execution::par***
- ***This will just work, (almost) always***

3rd Week preview, first algorithm: why this is complicated, why you want to use the standard library whenever possible

A simple stack:

- simple lock based concurrent variant
- ‘SIMPLE’ lock-free stack design....

Code can be downloaded at: https://github.com/zeta1999/TeachingDemoCPP/tree/master/demo/stack01

TECHNICAL DEPTH NOT IMPORTANT JUST RIGHT NOW. Goal of next slides is just to expose an important theme of the whole session:

Engineering trade-offs with concurrent code:

Simple code or high throughput or low latency or easily debuggable or ...

Technical depths will be explained ... at last ... in the end of week 3 (thread synchronization)

Adding lock-based concurrency: simple but...

ORIGINAL CODE

```
/** Non-thread-safe stack: Last In First Out */
template<typename T>
class stack {
  std::vector<T> contents;
  T *_contents = { nullptr };
  uint32_t index = { 0 };
  uint32_t size= { 0 };

public:
  [...]
  /** Add a COPY of t to the stack */
  inline void push(const T &t) {
                assert(_contents != nullptr);
                assert(index < size-1);
                _contents[index++] = t;
  }
```

LOCK-BASED: “minimal” changes

```
template<typename T>
class lstack {
  /** We wrap concurrent accesses... */
  stack<T> wrapped;
  /** With a lock guard on one mutex per stack... */
  std::mutex mx;
public:
  [...]
  /** Add a COPY of t to the stack */
  inline void push(const T&t) {
                // only one thread executes this code block at one time (per stack object)
                std::lock_guard<std::mutex> lck(mx);
                wrapped.push(t);
  }
```

(3rd week preview!)

Adding lock-free concurrency: NOT simple

ORIGINAL CODE

```
/** Non-thread-safe stack: Last In First Out */
template<typename T>
class stack {
  std::vector<T> contents;
  T *_contents = { nullptr };
  uint32_t index = { 0 };
  uint32_t size= { 0 };

public:
  [...]
  /** Add a COPY of t to the stack */
  inline void push(const T &t) {
                assert(_contents != nullptr);
                assert(index < size-1);
                _contents[index++] = t;
  }
```

LOCK-FREE: “many” changes

```
template<typename T>
class lfstack {
  /** Same as non-concurrent */
  std::vector<T> contents;
  /** Same as non-concurrent */
  T *_contents = { nullptr };
  /** Note the index is now atomic */
  std::atomic<uint32_t> index;
  /** Same as non-concurrent */
  uint32_t size= { 0 };
[...] // SIZEABLE PART
public:
[....]
 /** Add a COPY of t to the stack */
  inline void push(const T& t) {
                assert(_contents != nullptr);
                acquire_write_rights();
                // only one thread here ~ (write role)
                uint32_t index_copy = 0;
                do {
                  index_copy = index.load();
                  assert(read_index(index_copy) < size - 1);
                 _contents[read_index(clean_index(index_copy))] = t;
                  // incremented index has write flag set to ZERO
                } while(!std::atomic_compare_exchange_strong(&index,
                                                             &index_copy,
                                                                 increment_index(clean_index(index_copy))));
  }
```

(3rd week preview!)

Other issues

- Lock-free algorithms are typically difficult to compose (a lot more than lock-based concurrency)
- Some lock-free algorithm might be unsuitable to RT needs (latency distribution might have a fat tail, or just infrequent but of devastating consequence rare events)
- Difficult to debug: very few tools (to the best of our knowledge) and “of course”, classical instrumentation methodologies modify the running of the algorithms...
- Might have portability issues (say. Between, say x86_64 to ARM), due to the way relaxed barriers/acquire/release semantics are implemented.
- Performances might be lower than lock-based algorithms (so, do MEASURE/BENCHMARK performances)
- Complications with resize-able structures and allocation/reclamation: ABA problem, reclamation strategies (less so if using a GC, but is GC a good idea with RT and/or high performance code?)

(exact list of issues important here just by its length and depth at this stage)

(3rd week preview!)

Generic recommendations (esp. at intermediate level)

- Try to use standard library if possible
- Concurrent code is very intolerant of buffer overflows, leaks, etc.
- Try to use lock-based concurrency if possible (more tools to debug)
- If trying lock-free methods:
  - Make sure to model the CAS/atomic op loops. Analyse first, code second.
  - Measure performance! A badly tuned/inefficient lock-free algorithm might be less performant that a lock-based algorithm
  - Apply extra caution if RT requirements exist...

What will you achieve?
A highly concurrent chatbot!

Concurrent, high performance Chatbot

- Imagine, say, an internet bank user interface... => MANY users (but this is really one example)
- Can be developed at many levels of complexity...
  - Retrieval LSTM based (use ML/AI to match query to a set of predefined answers & behaviours)
  - Parse (possibly ML/AI based) then execute commands via non-ML AI approaches***
  - Use keywords & search infrastructure (succinct data structure for indexing?)
  - Do we want autocomplete/suggestions?
  - What GUI to use?
  - ...
- Many libraries can help
  - libtorch/C++
  - Tensorflow runtime
  - SDSL-lite
  - Networking libraries
  - ...

Contact us
“For now, there is no stupid question” M. G.

(a funny former boss)

Renoir42

https://zeta1999.github.io/renoir42/index.html

renoir42@yahoo.com