# Nov / 09 / 2020

## Multithreading Part lll and Functional Programming in C++

* Synchronization in C++: std::condition\_variable
	+ Updating thread:
* Acquire a std::mutex
* Perform update
* Call notify\_one() or notify\_all()
	+ Writing thread:
* Acquire the same std::mutex
* Call wait(), wait\_for() or wait\_until() with predicate
	+ Wait and notify:
* wait(lk, pred) : require lk to be std::unique\_lock<std::mutex>, since std::scoped\_lock<std::mutex> does not offer the lock and unlock operations required in wait
* For Read Write Lock: use std::condition\_variable\_any

A writer after finish can use std::condition\_variable\_any::notify\_all. All readers can then simultaneously hold the lock

* Higher level of multithread: in terms of tasks
	+ Communicating tasks: std::future<> and std::promise<>
* Enable transfer of value between threads without explicit using a lock
* Producer: use std::promise<> to set the value, or set\_exception() if the value cannot be produced
* Consumer: use std::future<> to get the value
	+ Using std::future<>
* T.get(): block the thread until value is available, or throw exception if producer sets the value as exception
* template<class Rep, class Period>

std::future\_status wait\_for(const std::chrono::duration<Rep,Period>& timeout\_duration) :

only block for the amount of [timeout\_duration] time

Returns : future\_status::deferred, future\_status::ready, or future\_status::timeout

* + Using std::promise<>
* std::future<R> get\_future(); --- Future object is obtained from promise object
* void set\_value(const R& value); --- Atomically store the value into shared state
* void set\_exception(std::exception\_ptr p); --- Indicating there won’t be any value in the future
	+ Communicating tasks: Managing futures and promises --- std::packaged\_task<R(Args ...)>
* A function wrapper, pass in functions
* template <class F> explicit pakcaged\_task(F&& f);
* std::future<R> get\_future(); --- adv: no worry about setting the value
* void operator() (ArgTypes … args);
* Example:

Thread1 will take the first half of the task and wait for the first thread to finish the computation. Thread1 will execute the task, and at f0.get() will return the value.

Double comp2(vector<double>& v)

{

 using Task\_type = double(double\*, double\*, double); // define type of task

 packaged\_task<TRask\_type> pt1{accum}; // accum is a function to pass to the package of task

 future <double> f0{pt0.get\_future()}; // get hold of pt0’s future

 double\* first = &v[0];

 thread t1 {move(pt0), first, first\_v.size()/2, 0} // start a thread for pt0

 Return f0.get(); // get the result

}

* + Communicating tasks: std::async
* Specify a task must run asynchronous, with no synchronization: how to run a program in sequential order? Using async
* For specialized parallel executions, C++ ‘s algorithm library offers execution policies, such as std::execution::seq, std::execution::par,...
* Example: starting an asynchronous task, while running other execution, so the thread will waiting at the asynchronous task, while executing other tasks

auto v0 = &v[0];

auto sz = v.size();

auto f0 = async(accum, v0, v0+ sz/4, 0.0)

..

Return f0.get() + f1.get() …

* + Thread pool
* Class ThreadPool{

Public:

threadPool(size\_t); // constructor takes [size\_t] number of threads

template<class F, class… Args>

auto enqueue(F&& f, Argos… args) // provide a function and arguments

 -> std:: future<typename std::result\_of<F(Args...)>::type>; // return type is future; compiler will figure out the function return type based on the input argos type

~ThreadPool(); // destructor: to manage threads, function for join, or detach

Private:

 std::vector <std::thread> workers; // keep track of threads

 std::queue< std::function<void()> > tasks; // tasks queue in FIFO order to execute

 std::mutex queue\_mutex;

std::condition \_varaible condition;

 bool stop;

}

Inline ThreadPool::ThreadPool(size\_t threads)

 : stop(false)

{

 for (size\_t i = 0; i<threads; ++i )

 workers.emplace\_back(

 [this] // use to access the element of the class

 {

 for(; ; )

 {

 std::function<void()> task;

 {

 std::unique\_lock<std::mutex> lock(this -> queue\_mutex);

 // if no task to be executed, they will be waiting on this conditional variable

 this->condition.wait(lock, [this]{return this->stop || !this->tasks.empty();}) // The condition is: no tasks needed to be done, whenever

we add a new tasks to the thread pool, we signal the threads so if there are threads waiting for new tasks could be notified and pick it up.

//If the condition is true, then it will just be a no-op, and it will proceed

 if(this -> stop && this->task.empty() )

 Return;

 Task = std::move(this -> tasks.front() );

 //…...

* Enqueue(): add new work item to the pool. Running the tasks asynchronously

// convert the function to another function that takes no arguments and return nothing,

 but want the future to be satisfied so that the client will be able to get the result.

template<class F, class… Args>

auto ThreadPool::enqueue(F&& f, Args&&... args)

 -> std::future<typename std::result\_of<F(Args...)>::type>

{

//….}

* Destructor:

Inline ThreadPool::~ThreadPool()

{

 {

 std::unique\_lock<std::mutex> lock(queue\_mutex);

 Stop = true;

}

condition.notify\_all(); // want all threads to be finished

for(std::thread &worker: workers)

 worker.join();

}

* Resize

// when the number of threads decreases

for(int i = oldNThreads -1; i >= nThreads; i--){

 \*this -> flags[i] = true; // this thread will finish

 this -> thread.detach(); // detach the threads that are no longer needed

 // let the threads running, exit from the distributor immediately

}

* Lazy Evaluation in C++
	+ Suggestion Book by Sagar: Chapter6 of Functional Programming in C++ by Ivan Cukic
	+ How to use lazy evaluation in C++: Write a function, and only evaluate it when needed; if it is a pure function, it can be cached and only return when requested
	+ Laziness in C++

Example.

// define a class lazy\_val with the following data members

template <typename F>

Class lazy\_val{

Private:

 F m\_computation;

 Mutable bool m\_cache\_initialized; // if there are results in the cache, if not need rerun

 Mutable decltype(m\_computation() ) m\_cache; // return type variable should be caching

 Mutable std::mutex m\_cache\_mutex;

Public;

 ...

}

// define computation

Operator const decltype(m\_computation() ) & () const

{

 std::unique\_lock<std::mutex> lock{m\_cache\_mutex};

 // check if the cache has the result computed

 if(!m\_cache\_initialized){

 // if not perform the computation

 m\_cache = m\_computation();

 m\_cache\_initialized = true;

}

Return m\_cache;

}

* + Std::call\_once --- not need to require the mutex; or change the variable

Every thread will wait until call\_once has been called, then m\_cache will have the result to return

template<typename F> class lazy\_val{

Private:

 F m\_computation;

 mutable decltype (m\_computation() ) m\_cache;

 Mutable std::once\_flag m\_value\_flag;

Public:

…

 Operator const decltype(m\_computation()) & () const{

 std::call\_once(m\_value\_flag, [this]){

 m\_cache = m\_computation();

}

return m\_cache;

}

}

* + Laziness as an optimization technique
* Adv: Only evaluated when needed & multiple invocation will only execute once
* Example1: displaying result page by page for a web query

Lazy quick sort: suppose only need 100 results from 1000 elements

 Only sort the items in the partition of first 100

* Example2: pruning recursive trees by caching function results

Fibonacci computation: maintain a cache of the computed fibonacci numbers, and latter when computing if has been computed then retrieve from the cache

auto fibmemo = make\_memorized\_r<unsigned int(unsigned int)>(

 [] (auto& fib, unsigned int n){

 Return n==0 ?0

 : n ==` ? 1

 : fib(n - 1) + fib(n - 2);

}

});

User need to pass in a function, with fibonacci as an argument

* Example3: Generalized memorization

// a function take in a function and returns an optimized function

template <typename Result, typename.. Args>

auto make\_memorized(Result (\*f) (Args))

// maintaining a map. Key is the arguments and value is the result

 std::map<std::tuple<Args...>, Result> cache;

 return [f, cache] (Args… args) mutable -> Result{

 const auto args\_tuple = std::make\_tuple(args...);

 const auto cached = cache.find(args\_tuple);

 If (cached == cache.end() ){

 auto result = f(args...);

 cache[args\_tuple] = result;

 return result;

}else{

 return cached->second;}

}

Above function doesn’t optimize for the recursive functions

* Expression template and lazy string

Example: concatenating multiple strings

// construct a new class object that stored all the value, and only concatenate when needed

template <typename... String> class lazy\_string\_concate\_helper;

// the last string and all the rest of the string, to construct a recursive data structure

template <typename LastString, typename… Strings>

Class lazy\_string\_concat\_helper<LastString, Strings...>{

…

 //build new Strings by Strings: return a higher level class object that contains the new string

 lazy\_string\_concat\_helper<std::string, LasString, Strings...>

operator+(const std::string& other) const{

 return lazy\_string\_concat\_helper<std::string, LastString, Strings...>(

 other,

\*this);

}