Multithreading in C++11

Threads, mutual exclusion and waiting

Klas Arvidsson

Software and systems (SaS)
Department of Computer and Information Science
Linköping University

LINKÖPING UNIVERSITY

LINKÖPING UNIVERSITY

std::thread

Functions passed to threads execute concurrently. Execution may be time-shared, simultaneous or both.

Constructor:

```
template< class Function, class... Args >
explicit thread( Function&& f, Args&&... args );
```

Selected members:

```
void join();
void detach();
bool joinable() const;
std::thread::id get_id() const;
static unsigned hardware_concurrency();
```

Example: receptionist and visitor

Thread function implementation

```
void receptionist(string name)
{
  cout << name << ": Welcome, how can I help you?" << endl;
  cout << name << ": Please enter, he's expecting you." << endl;
}
```

```
class Visitor
{
public:
  Visitor(string const& n) : name{n} {}
  void operator()() const
  {
    cout << name << ": Hi, I'm here to meet Mr X" << endl;
    cout << name << ": Thank you" << endl;
  }
private:
  string name;
};
```

Example: receptionist and visitor

Thread creation

```
#include <iostream>
#include <thread>
#include <chrono> // time constants

using namespace std;
using namespace std::chrono_literals; // time constants
```

```
int main()
{
  thread r{receptionist, "R"s};
  thread v{Visitor{"V"s}};
  thread f{[](){ cout << "F: Hi!" << endl; }};

  v.join();   // will wait for thread v to complete
  r.detach(); // makes you responsible ...
//  f.detach(); // terminate due to f not join'ed or detach'ed

  cout << "Main sleep" << endl;
  this_thread::sleep_for(2s); // pause main thread for 2 seconds
  cout << "Main done" << endl;
}
```

LINKÖPING UNIVERSITY

std::mutex

A basic building block for mutual exclusion. Variants include std::timed_mutex, std::recursive_mutex and (C++17) std::shared_mutex

Constructor:

```
constexpr mutex();
mutex( const mutex& ) = delete; // and also operator=
```

Selected members:

```
void lock();
bool try_lock();
void unlock();
```

std::shared_mutex (C++17)

A basic building block for mutual exclusion facilitating shared access to the resource. Shared access is commonly used for reading.

Constructor:

```
constexpr shared_mutex();
shared_mutex( const shared_mutex& ) = delete; // and also operator=
```

Selected members:

```
void lock();
bool try_lock();
void unlock();

void lock_shared();
bool try_lock_shared();
void unlock_shared();
```

std::lock_guard

Provides convenient RAII-style unlocking. Locks at construction and unlocks at destruction.

Constructor:

```
explicit lock_guard( mutex_type& m );
lock_guard( mutex_type& m, std::adopt_lock_t t );
lock_guard( const lock_guard& ) = delete; // and also operator=
```

std::unique_lock (C++11), std::shared_lock (C++14

Lock wrappers for movable ownership. An unique lock is required for use with std::condition_variable.

Features:

```
unique_lock();
unique_lock( unique_lock&& other );
explicit unique_lock( mutex_type& m );
unique_lock& operator=( unique_lock&& other );

shared_lock();
shared_lock( shared_lock&& other );
explicit shared_lock( mutex_type& m );
shared_lock& operator=( shared_lock&& other );
```

std::scoped_lock (C++17)

It locks all provided locks using a deadlock avoidance algorithm and with RAII-style unlocking.

Constructor:

```
explicit scoped_lock( MutexTypes&... m );
scoped_lock( MutexTypes&... m, std::adopt_lock_t t );
scoped_lock( const scoped_lock& ) = delete;
```

std::lock (C++11)

Function to lock all provided locks using a deadlock avoidance.

Use `std::scoped_lock` with `std::adopt_lock` on every lock after call to `std::lock` to get RAII-style unlocking.

```
template< class Lockable1, class Lockable2, class... LockableN >
void lock( Lockable1& lock1, Lockable2& lock2, LockableN&... lockn );
```

Example: Passing a mutex as reference parameter

Declaration and argument

```
int main()
{
  // Note: cout is thread safe on character level
  mutex cout_mutex;

  // references parameters have to be specified explicitly
  thread r(receptionist, ref(cout_mutex));
  thread v(Visitor{cout_mutex});

  r.join();
  v.join();

  cout << "Main done" << endl;

  return 0;
}
```

Example: Passing a mutex as reference parameter

Locking and unlocking

```
void receptionist(mutex& cout_mutex)
{
  cout_mutex.lock();
  cout << "R: Welcome, how can I help you?" << endl;
  cout_mutex.unlock();

  this_thread::yield(); // let other thread run

  lock_guard<mutex> lock(cout_mutex); // destructor auto unlock
  cout << "R: Please enter, he's expecting you." << endl;
}
```

Example: Passing a mutex as reference parameter

Using lock_guard for automatic unlock

```cpp
class Visitor
{
public:
  Visitor(mutex& cm) : cout_mutex{cm} {}

  void operator()()
  {
    cout_mutex.lock();
    cout << "V: Hi, I'm here to meet Mr X" << endl;
    cout_mutex.unlock();

    this_thread::yield(); // let other thread run

    lock_guard<mutex> lock(cout_mutex); // destructor auto unlock
    cout << "V: Thank you" << endl;
  }
private:
  mutex& cout_mutex;
};
```

Example: Separate block for std::lock_guard region

Using a separate block highlights the critical section

```
// some function
{
  foo();

  // critical section
  {
    lock_guard<mutex> lock(cout_mutex);
    cout << "After foo() but before bar()" << endl;
  }

  bar();
}
```

Example: Threads sharing cout

Each thread will print one line of text.

```
#include <iostream>
#include <vector>
#include <chrono>

#include <thread>
#include <mutex>

using namespace std;
using namespace std::chrono_literals;

int main()
{
  vector<string> v
  {
      "This line is not written in gibberish",
      "We want every line to be perfectly readable",
      "The quick brown fox jumps over lazy dog",
      "Lorem ipsum dolor sit amet"
  };
  mutex cout_mutex;
```

Example: Threads sharing cout

Thread implementation.

```
  auto printer = [&](int i)
    {
      string const& str = v.at(i);

      for (int j{}; j < 100; ++j)
      {
//          lock_guard<mutex> lock(cout_mutex);
        for (unsigned l{}; l < str.size(); ++l)
        {
          cout << str.at(l);
          this_thread::sleep_for(1us);
        }
        cout << endl;
      }
    };
```

Example: Threads sharing cout

Starting and joining our threads.

```
  vector<thread> pool;
  for ( unsigned i{}; i < v.size(); ++i )
  {
    pool.emplace_back(printer, i);
  }

  for ( auto && t : pool )
  {
    t.join();
  }
  cout << "Main done" << endl;

  return 0;
}
```

Example: Potential deadlock

Thread function

```
void deadlock(mutex& x, mutex& y)
{
  auto id = this_thread::get_id();

  lock_guard<mutex> lgx{x};
  cout << id << ": Have lock " << &x << endl;

  this_thread::yield(); // try to get bad luck here

  lock_guard<mutex> lgy{y};
  cout << id << ": Have lock " << &y << endl;

  cout << id << ": Doing stuff requiring both locks" << endl;
}
```

Example: Potential deadlock

Main: starting and joining our threads

```
int main()
{
  mutex A;
  mutex B;

  // references parameters have to be specified explicitly
  thread AB{deadlock, ref(A), ref(B)};
  thread BA{deadlock, ref(B), ref(A)};

  AB.join();
  BA.join();

  cout << "Main done" << endl;

  return 0;
}
```

Example: Potential deadlock

Deadlock avoidance

```
void no_deadlock(mutex& x, mutex& y)
{
  auto id = this_thread::get_id();

  // C++11, take locks
  lock(x, y);
  // And arrange for automatic unlocking
  lock_guard<mutex> lgx{x, adopt_lock};
  lock_guard<mutex> lgy{y, adopt_lock};

  // C++17
  // scoped_lock lock{x, y};
  cout << id << ": Have lock " << &x << " and " << &y << endl;

  cout << id << ": Doing stuff requiring both locks" << endl;
}
```

LINKÖPING UNIVERSITY

std::promise

Promise to deliver communication (or be done) in the future.

Constructor:

```
promise();
promise( promise&& other );
promise( const promise& other ) = delete;
```

Selected members:

```
std::future<T> get_future();
void set_value( const R& value );
void set_value();
void set_exception( std::exception_ptr p );
```

std::future

Waits for a promise to be fulfilled.

Constructor:

```
future();
future( future&& other );
future( const future& other ) = delete;
```

Selected members:

```
T get();
void wait() const;
```

Example: Using promise and future

Create promises and futures and move them

```
int main()
{
  promise<void> say_welcome;
  promise<string> say_errand;
  promise<void> reply;

  // You have to get the futures before you move the promise
  future<void> get_welcome = say_welcome.get_future();
  future<string> get_errand = say_errand.get_future();
  future<void> get_reply = reply.get_future();

  // You have to move promises and futures into the threads
  thread r(receptionist, move(say_welcome), move(get_errand), move(reply));
  thread v(visitor, move(get_welcome), move(say_errand), move(get_reply));

  // Wait for both threads to finish before continuing
  r.join();
  v.join();

  cout << "Main done" << endl;
}
```

Example: Using promise and future

Fulfill promise and wait for future

```
void receptionist(promise<void> say_welcome, future<string> errand,
                  promise<void> reply)
{
  cout << "R: Welcome, how can I help you?" << endl;
  say_welcome.set_value();

  string name = errand.get();
  cout << "R: Please enter, " << name << " is expecting you." << endl;
  reply.set_value();
}
```

```
void visitor(future<void> get_welcome, promise<string> tell_errand,
             future<void> get_reply)
{
  string name{"Mr X"};
  get_welcome.wait();
  cout << "V: Hi, I'm here to meet " << name << endl;
  tell_errand.set_value(name);
  get_reply.wait();
  cout << "V: Thank you" << endl;
}
```

LINKÖPING UNIVERSITY

std::condition_variable

Provides a way to wait for changes of a shared resource without blocking the resource lock.

Constructor:

```
condition_variable();
```

Selected members:

```
void notify_one();
void notify_all();
void wait( std::unique_lock<std::mutex>& lock );

template< class Predicate >
void wait( std::unique_lock<std::mutex>& lock, Predicate pred );
```

Example: Using a condition variable

Our worker thread

```
void worker(mt19937& die, int& done, mutex& m, condition_variable& change)
{
  uniform_int_distribution<int> roll(1,6);

  // just pretend to do some work...
  for ( int i{}; i < 100; ++i )
  {
    int n{roll(die)};
    for (int j{}; j < n; ++j)
      this_thread::sleep_for(1ms);

    lock_guard<mutex> lock(cout_mutex);
    cout << this_thread::get_id()
         << " iteration " << i
         << " slept for " << n << endl;
  }
  // message main thread that this thread is done
  unique_lock<mutex> done_mutex{m};
  --done;
  change.notify_one();
}
```

Example: Using a condition variable

Main: creating and detaching threads

```
int main()
{
  const int N{10};
  int done{N};
  random_device rdev;
  mt19937 die(rdev());

  mutex base_mutex{};
  condition_variable cond_change{};

  for (int i{}; i < N; ++i)
  {
    // if we do not need to keep track of threads we
    // can create and detach threads immediately
    thread(worker,
           ref(die),
           ref(done),
           ref(base_mutex),
           ref(cond_change)).detach();
  }
```

Example: using a condition variable

Main: finish when every thread is done

```
  // conditions require a std::unique_lock
  unique_lock<mutex> done_mutex{base_mutex};
  while ( done > 0 )
  {
    cout_mutex.lock();
    cout << "Main: still threads running!" << endl;
    cout_mutex.unlock();

    // we are holding the done_mutex and need to wait for another
    // thread to update the variable, but that thread can not lock the
    // done_mutex while we're holding it... condition_variables solve
    // this problem efficiently
    cond_change.wait(done_mutex);
  }
  done_mutex.unlock();

  // an option that would achieve the same as the loop above is to
  // keep track of all started threads in a vector and join them
  cout << "Main done" << endl;
}
```

LINKÖPING UNIVERSITY

std::packaged_task

Couple a task to deliver its result through a future, preparing it for asynchronous execution.

Class and constructor (compare to std::function):

```
template< class R, class ...Args >
class packaged_task<R(Args...)>;

template <class F>
explicit packaged_task( F&& f );
```

Selected members:

```
std::future<R> get_future();

void operator()( ArgTypes... args );
```

Example: Using a packaged task, setup

```
#include <iostream>
#include <vector>
#include <numeric>
#include <future>

#include "divider.h"

using namespace std;

using data = vector<int>;
using data_it = data::iterator;

int main()
{
  const auto thread_count{9};

  vector<int> v(100000000, 1);
  Divider<vector<int>> d{v, thread_count};
```

Example: Using a packaged task, divide work

```
vector<future<int>> partial_results;
for ( unsigned i{}; i < thread_count; ++i)
{
  // wrap our function in a future-aware object
  packaged_task<int(data_it, data_it, int)> worker(accumulate<data_it,int>);

  // get a handle to out future result
  partial_results.emplace_back( worker.get_future() );

  // execute our function in it's own thread
  thread{ move(worker), d.begin(i), d.end(i), 0 }.detach();
}
```

Example: Using a packaged task, fetch results

```
    cout << "Sum: "
         << accumulate(begin(partial_results), end(partial_results), 0,
                       [](int sum, future<int>& fut){ return sum + fut.get(); })
         << endl;

    return 0;
}
```

LINKÖPING UNIVERSITY

std::async

Prepare a function for asynchronous execution.

Function template:

```
template< class Function, class... Args >
// return type
std::future<std::invoke_result_t<std::decay_t<Function>, std::decay_t<Args>...>>
// function and arguments
async( std::launch policy, Function&& f, Args&&... args );
```

Policies:

```
std::launch::async enable asynchronous evaluation
std::launch::deferred enable lazy evaluation
```

Example: Using async, setup same as before

```cpp
#include <iostream>
#include <vector>
#include <numeric>
#include <future>

#include "divider.h"

using namespace std;

using data = vector<int>;
using data_it = data::iterator;

int main()
{
  const auto thread_count{9};

  vector<int> v(100000000, 1);
  Divider<vector<int>> d{v, thread_count};
```

Example: Using async, divide work

```
vector<future<int>> partial_results;
for ( unsigned i{}; i < thread_count; ++i)
{
  partial_results.emplace_back(
    // execute our function in it's own thread an get a handle to the future result
    // Note: always specify launch::async to avoid launch::deferred execution
    async<int(data_it, data_it, int)>(launch::async,
                                      accumulate, d.begin(i), d.end(i), 0)
    );
}
```

Example: Using async, fetch results same as before

```
  cout << "Sum: "
       << accumulate(begin(partial_results), end(partial_results), 0,
                     [](int sum, future<int>& fut){ return sum + fut.get(); })
       << endl;

  return 0;
}
```

LINKÖPING UNIVERSITY

std::execution

Execution policies let us specify sequential or parallell algorithm execution.

```
namespace execution {

class sequenced_policy;
class parallel_policy;

// execution policy objects:
inline constexpr sequenced_policy seq{ /*unspecified*/ };
inline constexpr parallel_policy par{ /*unspecified*/ };
inline constexpr parallel_unsequenced_policy par_unseq{ /*unspecified*/ };
}
```

Algorithm specification example:

```
template< class ExecutionPolicy, class RndIt, class Cmp >
void sort( ExecutionPolicy&& policy, RndIt first, RndIt last, Cmp comp );

template< class ExecutionPolicy, class FwdIt, class UnaryFunc2 >
void for_each( ExecutionPolicy&& policy, FwdIt f, FwdIt l, UnaryFunc2 f );
```

Example: Accumulate a vector, compare to previous

```cpp
#include <iostream>
#include <vector>
#include <numeric>

using namespace std;

using data = vector<int>;
using data_it = data::iterator;

int main()
{
  vector<int> v(100000000, 1);

  cout << "Sum: "
       << reduce(execution::par, begin(v), end(v), 0);
       << endl;

  return 0;
}
```

Example: Print independent vector values

Program template

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <numeric>  // <<-- iota
#include <algorithm>
#include <thread>
//#include <execution> // <<-- execution policies

using namespace std;

int main()
{
  vector<int> v(70);
  iota(begin(v), end(v), 1);
```

Example: Print independent vector values

Sequential execution policy

```
for_each(/*execution::seq,*/ begin(v), end(v), [](int i) {
    cout << setw(i) << 's' << endl;
  });
```

Example: Print independent vector values

Manual thread creation (for comparision)

```
const auto thread_count{4};
const auto size = v.size();
const auto chunk_size{ size/thread_count };
const auto remainder{ size%thread_count };

vector<thread> t;
auto b{ begin(v) };
for ( unsigned i{}; i < thread_count; ++i)
{
  auto e{ next(b, chunk_size + (i < remainder)) };
  t.emplace_back([](auto start, auto end){
      for (auto i{start}; i < end; ++i)
        cout << setw(*i) << '\\' << endl;
    }, b, e);
  b = e;
}

for ( auto && i : t )
  i.join();
```

Example: Print independent vector values

Parallell execution policy

Specified in C++17, but gcc support is still(2017-12-04) missing.

```
for_each(/*execution::par,*/ begin(v), end(v), [](int i) {
    cout << setw(i) << 'p' << endl;
  });
```

www.liu.se

LINKÖPING UNIVERSITY