Лекция 5.
Multithreading in C++

CS Club, Novosibirsk, 2019

Типы многозадачности

- Cooperative (совместная). Следующая задача выполняется только после того, как предыдущая явно отдала поток управления (lightweight threads, fibers).

- Preemptive (вытесняющая). Операционная система сама определяет когда забрать поток управления у задачи и отдать его другой задаче.

Многопоточность в C++

Threads management

- std::thread object
- threads utility (id, yield, sleep, …)
- thread_local (TLS)

Locking

- Mutual exclusion
- Lock management
- Condition variables

Tasks

- async
- future & promise

Parallel/vectorized algorithms (C++17)

Memory model

- models
- atomic template
- atomic_flag

C++20

- Executers
- Latches and barriers
- Coroutines*
- Transactional memory
- Task blocks
- jthread

Косвенно

- OpenMP*
- Exceptions
- Message/event loop
- Priorities

Многопоточность в C++

Threads management

- std::thread object
- threads utility (id, yield, sleep, …)
- thread_local (TLS)

Locking

- Mutual exclusion
- Lock management
- Condition variables

Tasks

- async
- future & promise

Parallel/vectorized algorithms (C++17)

Memory model

- models
- atomic template
- atomic_flag

C++20

- Executers
- Latches and barriers
- Coroutines*
- Transactional memory
- Task blocks
- jthread

Косвенно

- OpenMP*
- Exceptions
- Message/event loop
- Priorities

Многопоточность – это легко и эффективно. В теории

Часть 1. Параллельные алгоритмы

Параллельные алгоритмы (C++17)

```
1.  void parallel_run()
2.  {
3.      vector<double> data = {/*...*/};
4.      // before:
5.      sort(begin(data), end(data));
6.      // now we could run in parallel:
7.      sort(execution::par, begin(data), end(data));
8.  }
```

- Большая часть алгоритмов STL поддерживает теперь параллельный запуск с несколькими policy (стратегиями)

```
1.  execution::seq      // sequenced
2.  execution::par      // parallel
3.  execution::par_unseq// parallel + vectorized (SIMD)
4.  execution::unseq    // vectorized, since C++20
5.
6.  // 69 new functions + a few new
7.  sort, for_each, transform, tranform_reduce, ...
```

Низкая загрузка CPU

```
1.  std::transform(std::execution::par,
2.      vec.begin(), vec.end(), out.begin(),
3.      [](double v) { return v * 2.0; });
```

Operations	Vector Size	i7 4720 (4 Cores)	i7 8700 (6 Cores)
execution::seq	10k	0.002763	**0.001924**
execution::par	10k	0.009869	**0.008983**
openmp parallel for	10k	0.003158	**0.002246**
execution::seq	100k	0.051318	0.028872
execution::par	100k	0.043028	0.025664
openmp parallel for	100k	0.022501	0.009624
execution::seq	1000k	1.69508	**0.52419**
execution::par	1000k	1.65561	**0.359619**
openmp parallel for	1000k	1.50678	**0.344863**

https://www.bfilipek.com/2018/11/parallel-alg-perf.html

Высокая загрузка CPU

```
1.  std::transform(std::execution::par,
2.      vec.begin(), vec.end(), vecNormals.begin(),// input vectors
3.      vecFresnelTerms.begin(),                   // output term
4.      [](const glm::vec4& v, const glm::vec4& n)
5.      {
6.          return fresnel(v, n, 1.0f);
7.      });
```

Operation	Vector Size	i7 4720 (4 Cores)	i7 8700 (6 Cores)
execution::seq	10k	0.246722	**0.169383**
execution::par	10k	0.090794	**0.067048**
openmp parallel for	10k	0.049739	**0.029835**
execution::seq	100k	2.49722	1.69768
execution::par	100k	0.530157	0.283268
openmp parallel for	100k	0.495024	0.291609
execution::seq	1000k	25.0828	**16.9457**
execution::par	1000k	5.15235	**2.33768**
openmp parallel for	1000k	5.11801	**2.95908**

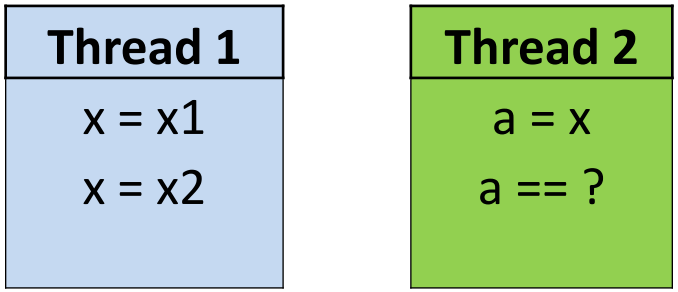
Соображения и рекомендации

- Параллельные алгоритмы делают больше работы чем последовательные
- Важно не только кол-во элементов, но и загрузка CPU каждой операцией (больше элементов и загрузки!)
- Лучше если меньше чтения памяти, больше CPU usage
- Правильно всегда проводить benchmark, чтобы оценить ускорение и целесообразность
- Начинайте с OpenMP, если он уместен

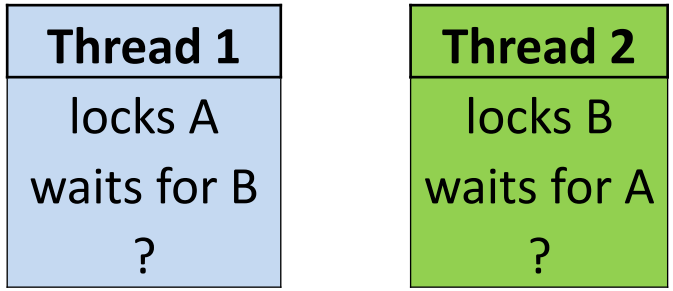
Часть 2. Потоки

Проблемные ситуации

- Race condition/data race (состояние гонки).



- Deadlock (взаимная блокировка).



- Livelock – блокировки как таковой нет, но крутимся в бессмысленном цикле.
- etc.

Нас не испугать dead lock’ами!

Создание потока

- Поток создается с помощью объекта std::thread (заголовочный файл <thread>).

```
1.  void big_calc(string how)
2.  {
3.      cout << "i\'m working " << how << "\n";
4.  }
5.
6.  int main()
7.  {
8.      thread th1{big_calc, "hard"};
9.      thread th2{big_calc, "24/7"};
10.
11.     // doing smth
12.
13.     th1.join();
14.     th2.join();
15.     return 0;
16. }
```

Объект потока std::thread

```
1.  // can be moved, not copied
2.  class thread
3.  {
4.      //...
5.      template< class Function, class... Args >
6.      explicit thread(Function&& func, Args&&... args);
7.
8.      // std::terminate if joinable
9.      ~thread();
10.
11.     bool                 joinable            () const noexcept;
12.     std::thread::id      get_id              () const;
13.     native_handle_type   native_handle       ();
14.     static unsigned      hardware_concurrency();
15.
16.     void join  (); // waits for thread func finish
17.     void detach(); // detaches thread object from system thread
18.     // ...
19. };
```

Вспомогательные функции

- Находятся в namespace `std::this_thread`
  - `yield` - отдает поток управления
  - `get_id` – возвращает `std::thread::id`
  - `sleep_for/sleep_until` – приостанавливает поток

```
1.  void thread_func()
2.  {
3.      lock_guard<mutex> lock(m);
4.
5.      if (task_queue.empty())
6.          std::this_thread::yield(); // Hmmm... Really?
7.      else
8.          //...
9.  }
```

Mutual exclusion

- mutex: обеспечивает базовые функции lock() и unlock() и не блокируемый метод try_lock()
- recursive_mutex: можно повторно захватить
- timed_mutex: в отличие от обычного мьютекса, имеет еще два метода: try_lock_for() и try_lock_until()
- recursive_timed_mutex: это комбинация timed_mutex и recursive_mutex
- shared_timed_mutex : предоставляет общий доступ нескольким потокам (читатели) или эксклюзивный одному потоку (писатель).

Lock management

- `lock_guard` – самый что ни на есть простой RAII держатеть mutex'а

- `unique_lock`
  - `try_lock[_for/_until]`
  - конструирование без или с тэгом: `defer_lock, try_to_lock, adopt_lock`
  - `owns_lock` проверка

- `shared_lock` – то же, что и `unique_lock`, но для `shared_timed_mutex`

Пример захвата mutex

- Чтобы избежать deadlock'а на нескольких mutex'ах используйте функцию `std::lock`

```
1.  void move_money(account& a1, account& a2, int amount)
2.  {
3.      unique_lock l1(a1.mut, defer_lock);
4.      unique_lock l2(a2.mut, defer_lock);
5.
6.      // avoids deadlocks
7.      std::lock(a1.mut, a2.mut);
8.
9.      if (a1.balance >= amount)
10.     {
11.         a1.balance -= amount;
12.         a2.balance += amount;
13.     }
14.     // ...
15. }
```

Exception catching

- Exception никаким образом не транслируется далее через границу потока.
- Что делать?

```
 1.  void thread_func()
 2.  {
 3.      try
 4.      {
 5.          // function body
 6.      }
 7.      catch (...)
 8.      {
 9.          // what to do with the exception?
10.      }
11.  }
```

Exception rethrow (1)

- Попытаемся его сперва поймать и сохранить!

```
1.  void thread_func()
2.  {
3.      try
4.      {
5.          // function body
6.      }
7.      catch (...)
8.      {
9.          std::exception_ptr p = current_exception();
10.         exceptions_queue.push_back(p);
11.     }
12. }
```

Exception rethrow (2)

```
1.  void invoker()
2.  {
3.      thread th(thread_func);
4.      //...
5.      th.join();
6.
7.      while(!exceptions_queue.empty())
8.      {
9.          try
10.         {
11.             auto p = exceptions_queue.top();
12.             exceptions_queue.pop();
13.             rethrow_exception(p);
14.         }
15.         catch(std::exception const& e)
16.         {
17.             cout << "FAILURE: " << e.what() << end;
18.         }
19.     }
20. }
```

Condition variables

- Блокируют поток(и) пока не будет получено уведомление от другого потока.
- До начала ожидания по condition variable (`wait`) блокируется `mutex`, а в самом wait – разблокируется.
- `wait` выходит, когда другой поток уведомляет condition variable (*либо* мифический *spurious wakeup*), *либо* timeout.
- Когда `wait` выходит, одновременно обратно блокируется mutex.

Что есть для condition variable в C++11?

- Классы

```
1. class condition_variable;
2. class condition_variable_any;
```

- Функции

```
1. void notify_one();
2. void notify_all();
3.
4. void wait( std::unique_lock<std::mutex>& l );
5. void wait( std::unique_lock<std::mutex>& l, Predicate pred );
6.
7. cv_status wait_for  (unique_lock<mutex>& l, const chrono::duration& t);
8. cv_status wait_until(unique_lock<mutex>& l, const chrono::time_point& t);
9 // the same + Predicate
```

Пример condition variable

```
1.  mutex                mt;
2.  condition_variable   cv;
3.  vector<char>         buf;
4.
5.  void sending_thread()
6.  {
7.    while (<some condition>)
8.    {
9.      unique_lock<mutex> l(mt);
10.     while (buf.empty())
11.       cv.wait(l);
12.
13.     send(buf.data(), buf.size());
14.   }
15. }
16. void on_frame(vector<char> const& data)
17. {
18.   {
19.     unique_lock<mutex> l(mt);
20.     buf.insert(buf.end(), data.begin(), data.end());
21.   }
22.   cv.notify_one();
23. }
```

Пример condition variable

```
1.  mutex                mt;
2.  condition_variable   cv;
3.  vector<char>         buf;
4.
5.  void sending_thread()
6.  {
7.    while (<some condition>)
8.    {
9.      unique_lock<mutex> l(mt);
10.     cv.wait(l, [](){ return !buf.empty(); });
11.
12.     send(buf.data(), buf.size());
13.   }
14. }
15. void on_frame(vector<char> const& data)
16. {
17.   {
18.     unique_lock<mutex> l(mt);
19.     buf.insert(buf.end(), data.begin(), data.end());
20.   }
21.   cv.notify_one();
22. }
```

Часть 3. Задачи

Отложенные вычисления

- Представим себе, что вычисления занимают длительное время.
- Закономерно, хочется передать такие вычисления другому потоку.
- Но результат желательно видеть в потоке, инициировавшем вычисления. *Pull model*
- Как это сделать, но при этом не создавать самим поток, защищать данные mutex’ом и т.д.

Future and Promise

- future позволяет дождаться вычисления результата

```
1.  void calc(promise<long> p)
2.  {
3.      long sum  = 0;
4.      long sign = 1;
5.      for (long i = 0; i < 100000000; ++i)
6.      { sum += i * sign; sign *= -1; }
7.
8.      p.set_value(sum);
9.  }
10.
11. int main()
12. {
13.     promise<long> p;
14.     future<long> f = p.get_future();
15.     thread t{calc, move(p)};
16.     ...
17.     std::cout << f.get() << '\n';
18.     t.join();
19. }
```

Packaged task and Future

- Как бы нам не вносить модификацию в саму функцию?

Packaged task and Future

- Как бы нам не вносить модификацию в саму функцию?

```
1.  long calc()
2.  {
3.      long sum  = 0;
4.      long sign = 1;
5.      for (long i = 0; i < 100000000; ++i)
6.      { sum += i * sign; sign *= -1; }
7.
8.      return sum;
9.  }
10.
11. int main()
12. {
13.     packaged_task<long> task{calc};
14.     future<long> f = task.get_future();
15.
16.     thread t{move(task)};
17.     ...
18.     cout << f.get() << endl;
19.     t.join();
20. }
```

async and future

- Можно ли проще, если не хочется управлять потоком самостоятельно?

```
1.  int main()
2.  {
3.    future<long> f = std::async(std::launch::async, calc);
4.    ...
5.    cout << f.get() << '\n';
6.  }
```

- Возможные стратегии запуска:
  - `std::launch::async` – в другом потоке
  - `std::launch::deferred` – в этом же потоке

future class

```
1.  template <class R>
2.  class future {
3.  public:
4.      /*constructors and assignment operators skipped*/
5.
6.      shared_future<R> share();
7.
8.      /*see description*/ get();
9.
10.     // functions to check state
11.     bool valid() const noexcept;
12.     void wait () const;
13.
14.     // template declaration are skipped
15.     future_status wait_for  (const chrono::duration& rel_time) const;
16.     future_status wait_until(const chrono::time_point& abs_time) const;
17. };
```

Часть 4. Очереди сообщений

Очередь событий/сообщений

- Хочу инициировать вычисления в параллельном потоке
- И получать нотификации с результатами вычислений в исходном (инициирующем) потоке. *Push model*
- Как?

Очередь событий/сообщений

```
1.  boost::asio::io_service io;
2.
3.  void some_long_calc_thread_1(function<void(long)> const& cb)
4.  {
5.      long res = 0;
6.      long sign = 1;
7.      for (long i = 0; i < 100000000; ++i)
8.      { res += sign * i; sign *= -1; }
9.
10.     io.post(bind(cb, res));
11. }
12.
13. void print_res_thread_0(long res)
14. { cout << res; }
15.
16. int main()
17. {
18.     thread th (some_long_calc_thread_1, print_res_thread_0);
19.     // do smth
20.     io.run();
21.     th.join();
22.     return 0;
23. }
```

Часть 5. Memory model

Давайте сделаем синглтон

- Защитим синглтон mutex'ом
- double-checked locking (anti-pattern)

```
1. singleton* instance() {
2.     if (ptr_ == nullptr) {
3.         lock_guard lock(mtx);
4.         if (ptr_ == nullptr) {
5.             ptr_ = new singleton();
6.         }
7.     }
8.     return ptr_;
9. }
```

Давайте сделаем синглтон

- Защитим синглтон mutex'ом
- double-checked locking (anti-pattern)

```
1. singleton* instance() {
2.     if (ptr_ == nullptr) {
3.         lock_guard lock(mtx);
4.         if (ptr_ == nullptr) {
5.             ptr_ = new singleton();
6.         }
7.     }
8.     return ptr_;
9. }
```

- Операция создания объекта и присваивания указателя не обязана быть атомарной

Давайте сделаем синглтон

- Компилятор может сделать создание примерно так, как написано ниже
- Теперь в этом коде кроется ошибка
- Исправляется это объявлением `std::atomic<singleton*> ptr_`

```
1.  singleton* instance() {
2.      if (ptr_ == nullptr) {
3.          lock_guard lock(mtx);
4.          if (ptr_ == nullptr) {
5.
6.              ptr_ = aligned_alloc
7.                  (alignof(singleton), sizeof(singleton));
8.
9.              /* creating thread may switch here... */
10.             ptr_ = new (ptr_) singleton;
11.         }
12.     }
13.     return ptr_;
14. }
```

Подготовим данные в другом потоке (lock free)

Thread 1

```
1.  uint64_t key{ 0 };
2.  volatile bool key_ready{ false };
3.
4.  void calculate_key()
5.  {
6.      secret_key = make_key();
7.      key_ready  = true;
8.  }
```

Thread 2

```
1.  void use_key()
2.  {
3.      if (key_ready)
4.          cout << secret_key;
5.  }
```

Компилятор может переставить обращения

Thread 1

```
1.  uint64_t key{ 0 };
2.  volatile bool key_ready{ false };
3.
4.  void calculate_key()
5.  {
6.      key_ready  = true;
7.      secret_key = make_key();
8.  }
```

Thread 2

```
1.  void use_key()
2.  {
3.      if (key_ready)
4.          cout << secret_key;
5.  }
```

- `volatile` никак не спасает от перестановки в C++, он вообще не для многопоточности

Вставим барьеры

Thread 1

```
1.  uint64_t key{ 0 };
2.  bool key_ready{ false };
3.
4.  void calculate_key()
5.  {
6.      secret_key = make_key();
7.      STORE ------- STORE
8.      key_ready  = true;
9.  }
```

Thread 2

```
1.  void use_key()
2.  {
3.      if (key_ready)
4.      LOAD ------- LOAD
5.          cout << secret_key;
6.  }
```

В C++11 определены модели памяти

```
1.  enum memory_order {
2.      memory_order_relaxed,
3.      memory_order_consume,
4.      memory_order_acquire, // LOAD---STORE ; LOAD---LOAD
5.      memory_order_release, // STORE---STORE; LOAD---STORE
6.      memory_order_acq_rel, // no STORE---LOAD; x86/64
7.      memory_order_seq_cst  // ALL
8.  };
```

```
1.  void do_smth_useful()
2.  {
3.      //...
4.      if (can_start) {
5.          ACQUIRE
6.          //...
7.          // any store/load stay between ACQUIRE&RELEASE
8.          //...
9.          RELEASE
10.     }
11. }
```

Подготовим данные в другом потоке (теперь правильно)

Thread 1

```
1.  uint64_t key{ 0 };
2.  atomic<bool> key_ready{ false };
3.
4.  void calculate_key()
5.  {
6.      secret_key = make_key();
7.      key_ready.store(true, memory_order_release);
8.  }
```

Thread 2

```
1.  void use_key()
2.  {
3.      if (key_ready.load(memory_order_acquire))
4.          cout << secret_key;
5.  }
```

Давайте сделаем правильный синглтон

```
1.  struct singleton
2.  {
3.      //...
4.
5.    static singleton* instance() {
6.        if (ptr_.load() == nullptr) {// memory_order_seq_cst
7.            lock_guard lock(mtx_);
8.            if (ptr_ == nullptr) {// memory_order_seq_cst
9.                // memory_order_seq_cst
10.               ptr_.store(new singleton());
11.           }
12.       }
13.       return ptr_;
14.   }
15.
16. private:
17.     static std::mutex mtx_;
18.     static std::atomic<singleton*> ptr_;
19. };
```

Здесь должно было наступить просветление!

Вопросы?