Лекция 13. Многопоточность. Часть 2

Что же еще есть в C++ для многопоточности?

- Condition variables
  - позволяют синхронизировать потоки по произошедшему событию
- Отложенные вычисления (promise, future, ...)
  - отложенное получение результата
- Очередь сообщений/событий
  - организация асинхронных вычислений
- Самостоятельно: atomic’и, memory_order, OpenMP

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. }
```

Отложенные вычисления

- Представим себе, что вычисления занимают длительное время.
- Закономерно, хочется передать такие вычисления другому потоку.
- Но результат желательно видеть в потоке, инициировавшем вычисления.
- Как это сделать?

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, ref(p)};
16.     ...
17.     std::cout << f.get() << '\n';
18.     f.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.     f.join();
20. }
```

Async and Future

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

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

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

Очередь событий/сообщений

- Хочу инициировать вычисления в параллельном потоке, а нотификацию с результатом получить в исходном (инициирующем) потоке.

- 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. }
```

Вопросы?