Параллельное программирование

C++. Thread Support Library.
Atomic Operations Library.

Развитие параллельного программирования в C++

- Posix threads
  - pthread_create
- Windows Threads
  - CreateThread
- OpenMPI
  - omp parallel
- **C++ Thread Support & Atomic Operations Libraries**
  - Нужна минимум
    - VS2012 (лучше VS2015)
    - GCC 4.8.1
    - Или 100 евро, чтобы купить just::thread

std::thread

- **std::thread** – стандартный класс потока
  - это не конкурент Windows и/или POSIX потокам
  - это обертка, которая внутри использует либо Windows-потоки, либо POSIX-потоки в зависимости от компилятора и платформы

Простой пример для std::thread

```
#include <thread>

void ThreadProc()
{
  printf("Inside thread = %d", std::this_thread::get_id());
}

std::thread t(ThreadProc);
…
t.join();
```

Обработка исключений

- Любое исключение – вылет, падение
- Привет исключениям по любому поводу, Boost!

```
void ThreadProc()
{
    try
    {
      // вычисления
    }
    catch (...)
    {
      // обработка исключения
    }
}
```

Копирование потоков std::thread

- Прямое копирование – ошибка компиляции

```
std::thread t(ThreadFunc);
t2 = t;
std::thread t3(t);
```

- `std::thread t2(std::move(t));` // t невалидно
- `std::thread& t3 = t2;` // валидно t2 и t3, но // это один и тот же // объект

Всегда надо join до пропадания std::thread из области видимости

```
#include <thread>

void ThreadProc()
{
  printf("Inside thread = %d", std::this_thread::get_id());
}

std::thread t(ThreadProc);
…
t.join();
```

Function objects

- Второй способ создания объектов std::thread

```
class FuncObject
{
public:
    void operator() (void)
     { cout << this_thread::get_id() << endl; }
};

FuncObject f;
std::thread t( f );
```

Лямбда-выражения

Closure semantics:
[ ]: none, [&]: by ref, [=]: by val, …

infer variable type

lambda arguments == parameters

return type…

```
auto lambda = [&] () -> int
{
    int sum = 0;
    for (int i=0; i<N; ++i)
        sum += A[i];
    return sum;
};
```

lambda expression = code + data

Потоки через лямбда-функции + передача параметров в потоки

```
for (int i=0; i<n; i++)
   z[i] = a * x[i] + y[i];
```

Parallel

```
auto code = [&](int start, int end) -> void
{
  for (int i = start; i < end; i++)
    z[i] = a * x[i] + y[i];
};


thread t1(code, 0   /*start*/, N/2 /*end*/);
thread t2(code, N/2 /*start*/, N   /*end*/);
```

Лямбда-функции vs обычные функции vs Function objects

- Разница в читаемости – на любителя
- При использовании лямбда-функций есть накладные расходы на создание объекта
- При использовании function objects тоже есть накладные расходы на создание объекта

Методы std::thread

- **joinable** – можно ли ожидать завершения потока (находимся ли мы в параллельном относительно этого потоке)
- **get_id** – возвращает ID потока
- **native_handle** – возвращает хендл потока (зависит от работающей реализации потоков)
- **hardware_concurency** – сколько потоков одновременно могут работать

Методы std::thread

- **join** – ожидать завершения потока
- **detach** – разрешить потоку работать вне зависимости от объекта std::thread
- **swap** – поменять два потока местами

- **std::swap** – работает с объектами типа std::thread

Методы std::this_thread

- **yield** – дать поработать другим потокам
- **get_id** – вернуть ID текущего потока
- **sleep_for** – «заснуть» на заданное время
- **sleep_until** – «заснуть» до заданного времени

```
std::chrono::milliseconds duration(2000);
std::this_thread::sleep_for(duration);
```

Недостатки std::thread

Как на std::thread сделать пул потоков (задач) ?

- Нет thread affinity
- Нет размера стека
  - для MSVC 2010 и выше не актуально
- Нет завершения потока
  - в pthread есть вполне приемлимая реализация
- Нет статуса работы потока
- Нет кода выхода потоковой функции

std::future

- **Future** – это высокоуровневая абстракция
  - Вы начинаете асинхронную операцию
  - Вы возвращаете хендл, чтобы ожидать результат
  - Создание потоков, ожидание выполнение, исключения и пр – делаются автоматически

std::async + std::future

START

```
#include <future>

std::future<int> f = std::async( []() -> int
  {
    int result = PerformLongRunningOperation();
    return result;
  }
);
.
.
```

lambda return type…

WAIT

```
try
{
  int x = f.get();   // wait if necessary, harvest result:
  cout << x << endl;
}
catch(exception &e)
{
  cout << "**Exception: " << e.what() << endl;
}
```

Где работает асинхронная операция?

- «Ленивое» исполнение в главном потоке
  - future<T> f1 =

    std::async(**std::launch::deferred**, []() -> T {...});
- Выполнение в отдельном потоке
  - future<T> f2 = std::async(**std::launch::async**, []() -> T {...});
- Пусть система решит, *она умнее*
  - future<T> f3 = std::async([]() -> T {...});

Wait For

- Аналог try-to-lock

```
std::future<int> f;
….
auto status =
   f.wait_for(std::chrono::milliseconds(10));
if (status == std::future_status::ready)
{
}
```

ready – результат готов
timeout – результат не готов
deferred – результат не посчитан, поскольку выбран «ленивый» подсчет

std::shared_future

- Аналог std::future, но позволяет копировать себя и позволяет ожидать себя нескольким потокам
- Например, чтобы можно было протащить future в несколько потоков и ждать его во всех них.
- Метод **share** объекта std::future возвращает эквивалентный std::shared_future

Методы std::future / std::shared_future

- **wait** – ждать результат
- **get** – получить результат
- **wait_for** – ожидать результат с таймаутом
- **wait_until** – ожидать результат максимум до заданного момента

std::packaged_task

```
std::packaged_task<int(int,int)> task([](int a, int b) { return
    std::pow(a, b); });
std::future<int> result = task.get_future();
task(2, 9);

std::packaged_task<int(int,int)> task(f);
std::future<int> result = task.get_future();
task();

std::packaged_task<int()> task(std::bind(f, 2, 11));
std::future<int> result = task.get_future();
task();
```

Зачем нужен std::packaged_task?

- Реиспользование
- Разная реализация
- Запуск на разных данных

Методы std::packaged_task

- **valid** – возвращает, установлена ли функция
- **swap** – меняет два packaged_task местами
- **get_future** – возвращает объект future
- **operator ()** – запускает функцию
- **reset** – сбрасывает результаты вычислений
- **make_ready_at_thread_exit** – запускает функцию, однако результат не будет известен до окончания работы текущего

Promises

- std::future дает возможность вернуть значение из потока **после** завершения потоковой функции
- std::promise это объект, который можно протащить в потоковую функцию, чтобы вернуть значение из потока **до** завершения потоковой функции

std::promise

```
void ThreadProc(std::promise<int>& promise)
{
    …
    promise.set_value(2));    //-- (3)
    …
}

std::promise<int> promise; //-- (1)
std::thread thread(ThreadProc, std::ref(promise)); //-- (2)
std::future<int> result(promise.get_future()); //-- (4)
printf("thread returns value = %d", result.get()) //-- (5)
```

Методы std::promise

- **operator ==** - можно копировать
- **swap** – обменять местами
- **set_value** – установить возвращаемое значение
- **set_value_at_thread_exit** – установить возвращаемое значение, но сделать его доступным только когда поток завершится
- **set_exception** – сохранить исключение, которое произошло
- **set_exception_at_thread_exit** – сохранить исключение, но сделать его доступным только после окончания работы потока

Locking

```
#include <mutex>

std::mutex  m;
int         sum = 0;
```

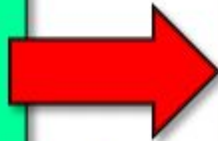
```
thread t1([&]()
  {
     int r = compute();
    m.lock();
     sum = sum + r;
    m.unlock();
  }
);
```

```
thread t2([&]()
  {
     int s = compute();
    m.lock();
     sum = sum + s;
    m.unlock();
  }
);
```

critical section

“Smart” locking

```
thread t([&]()
{
  int r = compute();
  m.lock();
  sum += r;
  m.unlock();
});
```



should be written as…

```
thread t([&]()
  {
    int r = compute();
    {
      lock_guard<mutex> lg(m);
      sum += r;
    }
  });
```

Locks m in constructor

Unlocks m in destructor

Методы std::mutex

- **lock** – захватить мьютекс
- **unlock** – освободить мьютекс
- **try_lock** – попробовать захватить мьютекс с таймаутом 0 секунд и возвратить, успешно или нет
- **native_handle** – хендл мьютекса (зависит от реализации)

Другие виды мьютексов

- **std::timed_mutex** – мьютекс, который можно попробовать захватить с ненулевым таймаутом
- **std::recursive_mutex** – мьютекс, который может быть многократно захвачен одним и тем же потоком
- **std::recursive_timed_mutex** – смесь *std::timed_mutex* и *std::recursive_mutex*

Методы для timed мьютексов

- **try_lock_for** – попытка захватить мьютекс с заданным таймаутом и возвратом успешности операции
- **try_lock_until** – попытка захватить мьютекс максимум до заданного времени и возвратом успешности операции

std::shared_timed_mutex (C++ 14)

- Объект, который позволяет эксклюзивно захватывать мьютекс и неэксклюзивно.
- Если мьютекс захвачен эксклюзивно, то неэксклюзивно захватить его нельзя (ожидание). Обратное верно.
- Неэксклюзивный захват может быть из нескольких потоков одновременно

std::shared_timed_mutex

- Зачем нужно?
  - На чтение защищенный ресурс можно открыть из нескольких потоков без возникновения проблем
  - На запись можно открыть только одному потоку, причем чтобы в этот момент никто не читал – чтобы проблем не было

std::shared_timed_mutex

- Эксклюзивный доступ
  - **lock**
  - **try_lock**
  - **try_lock_for**
  - **try_lock_until**
  - **unlock**

std::shared_timed_mutex

- Неэксклюзиный доступ
  - lock_shared
  - try_shared_lock
  - try_shared_lock_for
  - try_shared_lock_until
  - unlock_shared

Общие алгоритмы захвата

Deadlock avoidance algorithm and exception handling

- std::lock

  std::lock(mutex1, mutex2, …, mutexN);

- std::try_lock – с нулевым таймаутом

  std::try_lock(mutex1, mutex2, …, mutexN);

std::call_once & std::once_flag

Запуск функции только 1 раз (на все потоки 1 раз). Уникальную функцию идентифицирует объект std::once_flag

```
std::once_flag flag;

void do_once()
{
    std::call_once(flag, []() { printf(“called once”); });
}

std::thread t1(do_once);
std::thread t2(do_once);
t1.join(); t2.join();
```

Умные указатели для примитивов синхронизации

Для обоих нельзя копировать, можно переносить. Различия

- **std::unique_lock** – обертка для эксклюзивного доступа
- **std::shared_lock** **(C++ 14)** – обертка для неэксклюзивного доступа

Методы std::shared_lock / std::unique_lock

- **operator =** разблокирует текущий мьютекс и становится оберткой над новым
- **lock**
- **try_lock**
- **try_lock_for**
- **try_lock_until**
- **unlock**

Методы std::shared_lock / std::unique_lock

- **swap** – обменять примитив синхронизации с другим объектом ?_lock
- **release** – отсоединить текущий примитив синхронизации без unlock
- **mutex** – возвращает ссылку на текущий примитив синхронизации
- **owns_lock** – возвращает true, если управляет примитивом синхронизации

Стратегии захвата примитива синхронизации в конструкторе

```
std::lock_guard<std::mutex> lock1(m1,
    std::adopt_lock);
std::lock_guard<std::mutex> lock2(m2,
    std::defer_lock);
```

- **std::defer_lock** – не захватывать мьютекс
- **std::try_to_lock** – попробовать захватить с нулевым таймаутом
- **std::adopt_lock** – считать, что текущий поток уже захватил мьютекс

Событие: std::condition_variable

- **notify_one** – уведомить о событии 1 поток
- **notify_all** – уведомить о событии все потоки
- **wait** – ждать события
- **wait_for** – ждать события с таймаутом
- **wait_until** – ждать события не дольше, чем до заданного времени
- **native_handle** – вернуть хендл события (зависит от реализации)

std::atomic

```
#include <atomic>

std::atomic<int> count(0);
```

```
thread t1([&]()
 {
   count++;
 });
```

```
thread t2([&]()
 {
   count++;
 });
```

```
thread t3([&]()
 {
   count = count + 1;
 });
```

not safe...

std::atomic<T>

- Шаблонный тип данных для цифровых переменных (char, short, int, int64, etc) и указателей
- Почти всегда lock-free, а если и не lock-free, то не требует написания lock-ов вами.
- Главное отличие – гонки данных исключены.
- Зато возможные операции крайне ограничены.

std::atomic

- **operator =** приравнивает один атомик другому
- **is_lock_free** – возвращает, является ли реализация для этого типа данных lock free
- **store** – загружает в атомик новое значение
- **load** – получает из атомика значение
- **exchange** – заменяет значение атомика и возвращает прошлое значение

std::atomic

- **fetch_add, fetch_sub, fetch_and, fetch_or, fetch_xor** – выполняет сложение, вычитание, логические И, ИЛИ, XOR и возвращает предыдущее значение
  - **operator++**
  - **operator++(int)**
  - **operator--**
  - **operator--(int)**
  - **operator+=**
  - **operator-=**
  - **operator&=**
  - **operator|=**
  - **operator^=**

std::atomic_flag

- **operator =** - присвоение
- **clear** – сброс флага в false
- **test_and_set** – устанавливает флаг в true и возвращает предыдущее значение

Что еще есть в C++ 11 Atomic Operations Library?

- Дублирование всех методов внешними операциями (.swap -> std::swap)
- Compare And Swap (CAS)
- И еще много всего

для тех, кто слишком хорошо понимает, что делает

Общие впечатления

- Шаг вперед по адекватности и однообразию
- Местами шаг назад. Сравните

```
Sleep(100)
```

и

```
std::chrono::milliseconds duration(100);
std::this_thread::sleep_for(duration);
```

Общие впечатления

- Впервые потоки, примитивы синхронизации стандартизированы
- Некоторые устоявшиеся термины и подходы исключены (семафор; код, возвращаемый потоковой функцией; принудительное завершение потока; thread affinity; размер стека)
  - Некоторые совсем
  - Для некоторых непривычные аналоги

Общие впечатления

- В целом функциональности много, есть полезные нововведения, которые позволяют свести необходимость программировать синхронизацию к нулю.
- Однако, с другой стороны отсутствие определенной функциональности, мягко говоря, смущает.

Вопросы