Джефри Рихтер:
«Не нужно плодить потоки, потому что переключение контекстов - вещь дорогая»

THREAD POOL

Проблемы нерационального использования потоков

Если для решения часто возникающих небольших заданий для каждого задания запускается отдельный поток:

- поток создается непосредственно при поступлении задания и разрушается по завершении его обработки => порождение и уничтожение потока приводят к **значительным издержкам**
- требуется «**вручную**» задавать/определять количество одновременно работающих потоков
- для некоторых задач требуется организовать досрочное завершение потока (в классе std::thread не реализовано)

Шаблон проектирования «пул потоков»

- позволяет создавать разумное количество потоков
- и многократно их использовать, пока существует пул
- при уничтожении самого пула потоки также удаляются.

В результате экономятся ресурсы, которые до этого шли на создание, переключение и уничтожение потоков.

Идея пула потоков:

- создаем N потоков и помещаем их в хранилище (пул).
- по мере возникновения новой задачи «достаем» из хранилища готовый поток и отдаем ему задачу на выполнение (при этом не происходит каждый раз создание/уничтожение => при частых обращениях выигрыш)
- как только поток отрабатывает отведенную ему задачу - он высвобождается => можно использовать его для выполнения следующей задачи.

Простейшая реализация пула потоков

Сущности:

- рабочий поток,
- пул потоков,
- задание
- очередь заданий

В данной реализации не предусмотрены:

- получение результатов выполнения задания
- синхронизация с завершением задания
- генерация исключений заданиями

```
class thread_pool {
  std::mutex m; //для безопасной многопоточной работы с очередью
  std::queue<std::function<void()>> tasks; //извне не
        используется, а в методах обеспечивает корректный доступ
        к разделяемым данным посредством мьютекса
  std::vector<std::thread> threads; //совокупность рабочих
потоков

  void task_thread(); //потоковая функция

public:
  thread_pool();
  ~thread_pool();
  void add_task(параметры);
};
```

```
void thread_pool::task_thread()
{
  while (true){
        std::function<void()> task;
        m.lock();
        if (tasks.empty()) {
              m.unlock();
              std::this_thread::yield(); //отдыхаем
        }else {
              task = tasks.front();
              tasks.pop();
              m.unlock();

        }
              if (task) task();
        }

}
```

```
thread_pool::thread_pool() {
   size_t nThreads = определяем количество потоков в пуле
   //Запускаем потоки:
   for (size_t i = 0; i < nThreads; i++){
   threads.emplace_back(&thread_pool::task_thread, this);
//может быть сгенерировано исключение => по-хорошему нужно обработать
   }
}

thread_pool:: ~thread_pool(){
        ???
}
```

Продолжение

```
void thread_pool:: add_task(параметры)
{
        m.lock();
        tasks.push(функтор_с_параметрами);
        m.unlock();
}
```

#include <conditional_variable>

УСЛОВНЫЕ ПЕРЕМЕННЫЕ

Если один поток должен дождаться выполнения условия другим потоком:

bool ReadyFlag = false;
std::mutex m;

```
void thread1()
{
 m.lock();
//подготовка данных
 ReadyFlag = true;
  m.unlock();
//…
}
```

```
void thread2(){
 std::unique_lock <std::mutex > l(m);
 while(!ReadyFlag){
    l.unlock();
    std::this_thread::sleep_for(100ms);
    l.lock();
  }
//обработка данных
}
```

Проблемы:

- флаг проверяется независимо от того, произошло событие или нет
- в среднем поток будет ждать duration/2
- ресурсы на unlock()/lock()

=> это неэффективный вариант

=> использование специальных средств для решения таких задач – conditional variables!

Цель использования условных переменных:

- когда нужно не просто избежать гонки (mutex), а синхронизировать параллельное выполнение задач
- требуется НЕ однократная передача результата от одного потока другому (future - однократная синхронизация), а многократное получение результатов от разных потоков
- => условные переменные, которые можно использовать для многократной синхронизации при обмене данными с несколькими потоками

Реализации условных переменных

std::condition_variable обеспечивает синхронизацию только посредством мьютекса

std::condition_variable_any обеспечивает синхронизацию посредством любого «мьютексоподобного» объекта => может потребовать дополнительных расходов по памяти, производительности, ресурсов ОС

std::condition_variable
std::condition_variable_any

это объекты синхронизации, предназначенные для блокировки одного потока, пока

- он не будет оповещен о наступлении некоего события из другого потока
- или не истечет заданный таймаут
- или не произойдет ложное пробуждение (spurious wakeup)

Отправка/получение оповещения

Поток, формирующий условие:

- **notify_one**() – сообщить одному потоку
- **notify_all**() – сообщить всем потокам

- Поток, ожидающий выполнения условия:
- **wait**() – ожидание выполнения условия
- **wait_for**(), **wait_until**()

Важно!

- в отличие от мьютекса условная переменная позволяет **одному** из ожидающих или **всем** ожидающим потокам продолжить выполнение
- для реализации ожидания на условной переменной все равно необходим мьютекс
- «ложное срабатывание» == возврат управления потоку не всегда означает выполнение условия => необходимо дополнительно проверить выполнение условия

Ложное (spurious) пробуждение

- когда wait завершается, без участия notify_one() или notify_all()

То есть проверка условной переменной может сработать даже если условная переменная не осуществила уведомления => дополнительная проверка!

A.Williams: “Ложные срабатывания невозможно предсказать: с точки зрения пользователя они являются совершенно случайными. Однако они часто происходят, когда библиотека потоков не может гарантировать, что ожидающий поток не пропустит уведомления. Поскольку пропущенное уведомление делает условную переменную бесполезной, библиотека потоков активизирует поток, чтобы не рисковать”

void **wait**(std::unique_lock<std::mutex>& lock);

- при первом вызове ,если мьютекс занят, освобождает ассоциированный мьютекс (это позволяет другим потокам изменять защищенные данные во время ожидания)
- добавляет этот поток в список ожидающих потоков во внутреннюю структуру данных условной переменной и блокирует вызвавший wait() поток
- при вызове для условной переменной notify_…() анализируется список и пробуждается один из или все ожидающие потоки, управление возвращается функции wait(), в которой снова захватывается мьютекс и выполнение продолжается

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

- используется для защиты от ложных пробуждений, так как
- проверяет дополнительное условие, и если условие==false, то снова освобождает мьютекс и переводит поток в состояние ожидания
- эквивалентно:

```
while (!pred()) {
    wait(lock);
}
```

Важно!

- на момент вызова wait() мьютекс должен быть захвачен! Иначе – неопределенное поведение
- во всех **ожидающих** потоках должен использоваться один и тот же мьютекс
- начиная с C++14 wait() не генерирует исключений

wait_for(), wait_until()

возврат управления и блокировка мьютекса происходит:

- если вызвана notify_...()
- сработало ложное пробуждение
- истек timeout - wait_for().
- или наступил заданный момент времени - wait_until()

Причину возврата можно узнать посредством возвращаемого значения - enum class **cv_status**; (значения: timeout, no_timeout)

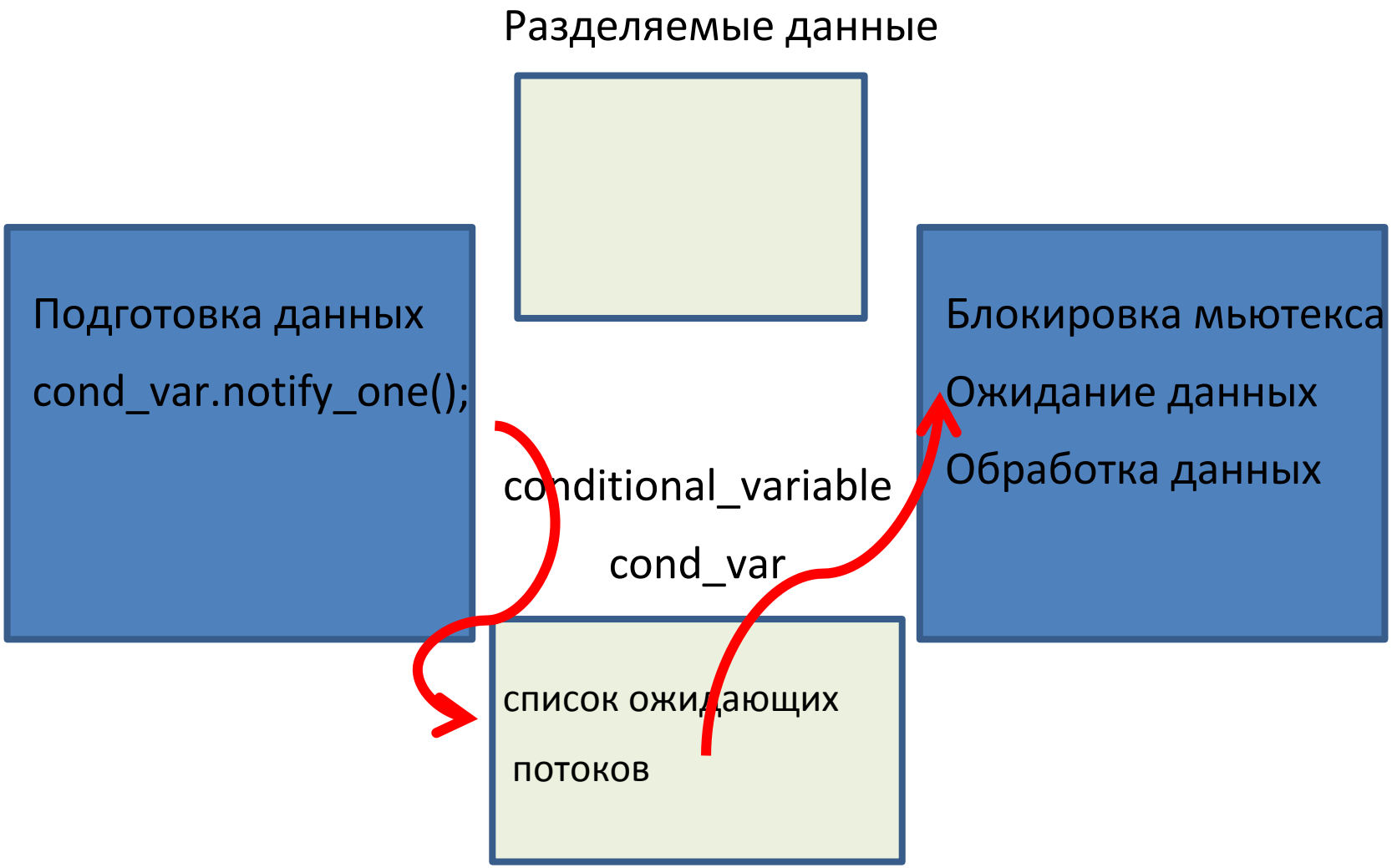
notify_one(), notify_all()

при вызове для условной переменной notify_…() анализируется список и пробуждается:

- один из ожидающих потоков
- или все ожидающие потоки

=> управление возвращается соответствующей функции wait()

Иллюстрация



Ожидание условия с помощью условной переменной в **одном** потоке:

int shared = 0;
std::condition_variable cv;

```
void thread_Write() {
//подготовка данных
shared = 33;
//уведомление о готовности
cv.notify_one();

}
```

```
void thread_Read (){
std::mutex   local_m;

std::unique_lock <std::mutex > l(local_m);

 cv.wait(l);

//обработка данных
}
```

Ожидание условия с помощью условной переменной в **одном** потоке. Защита от ложных пробуждений

int shared = 0;
std::condition_variable cv;

```
void thread_Write() {
//подготовка данных
shared = 33;
//уведомление о готовности
cv.notify_one();

}
```

```
void thread_Read (){
std::mutex local_m;

std::unique_lock <std::mutex > l(local_m);

cv.wait(l, []{ return shared == 33; });

//обработка данных
}
```

Ожидание условия с помощью условной переменной в **нескольких** потоках:

std::mutex m;
std::conditional_varible cv;
int shared = 0;

```
void thread_Write() {
//подготовка данных
shared = 33; //гонка!
//уведомление о
готовности
cv.notify_all();

}
```

```
void thread_Read(){
 std::unique_lock <std::mutex > l(m);
 cv.wait(l,  [] { return shared == 33; });
  …
}
```

Уведомление с помощью условной переменной в **нескольких** потоках:

std::mutex m;
std::conditional_varible cv;
int shared = 0;

```
void thread_Write() {
std::lock_guard<std::mutex>   l(m);
//подготовка данных
shared = 33;
//уведомление о готовности
cv.notify_all();

}
```

```
void thread_Read(){
std::unique_lock <std::mutex> l(m);
cv.wait(l,
            [] { return shared == 33; });
 …
}
```

Глобальная функция std::notify_all_at_thread_exit()

void **notify_all_at_thread_exit**(
    std::condition_variable& cond,
    std::unique_lock<std::mutex> lk);

Пробуждает все ожидающие потоки (cond.wait()) **при завершении** текущего потока. Эквивалентно:

lk.unlock();

cond.notify_all();

Важно! Действия выполняются гарантированно **после** вызова деструкторов всех потоко-локальных объектов (thread local storage duration)

Пример std::notify_all_at_thread_exit

std::mutex m;
std::conditional_varible cv;
int shared = 0;

```
void thread_Write() {
std::unique_lock<std::mutex>   l(m);

//использование thread_locals


//уведомление о готовности выдать только
после завершения потока
std::notify_all_at_thread_exit(cv,
            std::move(l));
…
}//1. деструкторы для thread_locals, 2.
unlock mutex, 3. notify_all()
```

```
void thread_Read(){
std::unique_lock <std::mutex>
                l(m);
cv.wait(l,  <проверка усл.>);
…
}
```

Пример std::condition_variable

```
std::vector<int> data;
std::condition_variable cv;
std::mutex m;

void thread_func1() {
  std::lock_guard<std::mutex> lock(m);
  data.push_back(10);
  cv.notify_one();
}
void thread_func2() {
  std::unique_lock<std::mutex> lock(m);
  cv.wait( lock, []() { return !data.empty(); } );
  std::cout << data.back() << std::endl;
}
```

std::condition_variable::native_handle()

native_handle_type native_handle();

В ОС Windows условные переменные на системном уровне поддерживаются объектом синхронизации **CONDITION_VARIABLE**

Системные функции для работы с CONDITION_VARIABLE:
InitializeConditionVariable(), SleepConditionVariableSRW (), WakeConditionVariable()

Высокоуровневые средства запуска и взаимодействия потоков. Асинхронное программирование. Ожидание одноразовых результатов

#include <future>

STD::ASYNC()

STD::FUTURE

STD::SHARED_FUTURE

Результат работы потоковой функции посредством std::thread?

в качестве одного (нескольких) параметра в потоковую функцию можно отправить адрес, по которому функция «положит» сформированный ею результат

Проблемы:

- при ожидании завершения запущенного потока???
- если detach()???

Задача:

Требуется запуск длительной операции и получение результата. При этом результат понадобится позже (а может быть и вообще не понадобится…)

Варианты:

- вызвать синхронно =>
  ‘+’ - результат гарантированно сформирован после вызова
  ‘- ‘ – время, потраченное на вычисление, можно было бы потратить на выполнение какой-нибудь другой полезной работы, не требующей результата

Проблемы получения результата **detached** потока:

- как узнать о том, что результат сформирован?
- как обеспечить существование объекта, адрес которого передан в поток?

Асинхронное программирование

стиль программирования, при котором :

- «тяжеловесные» задачи исполняются в detached дочерних (фоновых) потоках,
- и результат выполнения которых может быть получен родительским потоком, когда он того пожелает (при условии, что результат доступен)

шаблон функции std::async()

- позволяет **синхронно** или **асинхронно** запустить задачу (потенциально в отдельном потоке)
- передать ей любое количество параметров (аналогично std::thread)
- и получить возвращаемое потоком значение с помощью шаблона std::future (когда оно будет сформировано)

Формы async() – C++14

template< class Function, class... Args>

std::**future**<std::result_of_t<std::decay_t<Function>(std::decay_t<Args>...)>>

async(**Function&& f, Args&&... args**);

template< class Function, class... Args >

std::**future**<std::result_of_t<std::decay_t<Function>(std::decay_t<Args>...)>>

async(**std::launch policy, Function&& f,**
Args&&... args);

Специфика **std::launch** :

- По умолчанию std::launch::**deferred |** std::launch::**async** реализация решает сама: запускать новый поток или выполнять синхронно
- std::launch::**deferred** – отложить вызов функции до вызова методов wait() или get() объекта future (*lazy evaluation*)
- std::launch::**async** – запуск функции в отдельном потоке

Для асинхронного взаимодействия и обмена данными thread support library предоставляет:

- **future**s – средства для однократного
  - получения возвращаемых потоками значений
  - и обработки исключений при асинхронном выполнении

Будущие результаты (future)

```
std::future<> & std::shared_future<>
```

Важно!

- Будущие результаты сами по себе не обеспечивают синхронизацию доступа к разделяемым данным
- => если несколько потоков используют один и тот же объект future,
  - то они сами должны синхронизировать доступ с помощью мьютекса или других средств синхронизации
  - или нужно позволить нескольким потокам пользоваться одним и тем же объектом **std::shared_future** или каждому потоку работать со своей копией - **std::shared_future** без дополнительной синхронизации

Шаблон класса std::future:

- формально является **оберткой** для значения любого типа, вычисление или получение которого происходит отложено (то есть в неизвестном будущем)
- фактически поддерживает механизм однократного уведомления, является получателем будущего значения
- сама по себе обертка пассивна, она просто предоставляет доступ к некоторому **разделяемому состоянию**, которое состоит из 2-х частей: собственно интересующие **данные** + **флаг готовности** (как только значение вычислено, устанавливается флаг)
- для многих задач менее ресурсоемко, чем condition variable + mutex

Шаблон std::future

предоставляет механизм для однократного получения результата асинхронной операции, инициированной: **std::async**(), std::packaged_task(), std::promise()

- готовность результата посредством методов std::future:
  - **wait()** - блокирует вызвавший поток до получения результата
  - **wait_for()** – ждет завершения или истечения timeout-а
  - **wait_until()** - ждет завершения или наступления момента
  - **get() –** для ожидания вызывает wait() и возвращает результат

std::future::**get()**

- **T get()**; //генеральный шаблон (возвращаемое значение формируется посредством std::move()) =>
  повторный вызов == неопределенное поведение
- **T& get()**;//для специализации future<T&>
- **void get()**; // для специализации future<void>

Важно!

- Любой из вариантов дожидается результата
- Если было сохранено исключение, оно заново генерируется

std::future::**get()**

При вызове get() могут быть три ситуации:

- если выполнение функции происходило в отдельном потоке и уже закончилось (или результат уже сформирован) – сразу получаем результат
- если выполнение функции происходит в отдельном потоке и еще не закончилось (или результат еще не сформирован) – родительский поток блокируется
- если выполнение функции еще не начиналось, то вызов происходит синхронно

Важно: вторичный вызов get() – неопределенное поведение!

Пример std::async(). Поведение по умолчанию

```
int sum(const std::vector<int>& v){
    int res = 0;
    for (auto i:v) { res += i;}
    return res;
}

int main(){
    std::vector<int> v = { 1, 2, 3, 4 };
    std::future<int> f = std::async(sum, std::cref(v));
    //…какие-то вычисления
    std::cout<<f.get(); //дожидаемся окончания,
                        получаем результат
}
```

Пример std::async() Явное задание условий вызова:

```
int sum(const std::vector<int>& v){
    int res = 0;
    for (auto i:v) { res += i;}
    return res;
}

int main(){
    std::vector<int> v = { 1, 2, 3, 4 };
    std::future<int> f = std::async(std::launch::async, sum,
                                    std::cref(v));
    //…какие-то вычисления
    std::cout<<f.get(); //дожидаемся окончания,
                        получаем результат
}
```

Пример std::async() Явное задание условий вызова:

```
int sum(const std::vector<int>& v){
    int res = 0;
    for (auto i:v) { res += i;}
    return res;
}

int main(){
    std::vector<int> v = { 1, 2, 3, 4 };
    std::future<int> f = std::async( std::launch::deferred, sum,
                                     std::cref(v));
    //…какие-то вычисления
    std::cout<<f.get(); //синхронный вызов + прием возвращаемого
                        значения
}
```

Специфика: если функция ничего не возвращает

```
void func(void);


int main(){
   std::future<void> f = std::async( func);
  //…какие-то вычисления
    f.get();  //эквивалентно f.wait();
}
```

Пример использования wait_for()

```
while
(f.wait_for(1s)==std::future_status::timeout) {
    // выполняем другую работу, не требующую результата
      }
 //работаем с результатом
```

std::future::valid()

```
int func(int);
int main(){
   int res=0;
   std::future<int> f1 = std::async( func);
   bool b = f1.valid(); //true
   std::future<int> f2 = std::move( f1);
   b=f1.valid(); //false
   if(f2.valid()){  res = f2.get();} //true
   b=f2.valid(); //false
}
```

Сохранение исключения в объекте future

Если функция, вызванная посредством async(), генерирует исключение,

- это исключение сохраняется в объекте future вместо значения (результата)
- вызов get() повторно возбуждает сохраненное исключение (при этом стандарт не оговаривает: в объекте future создается копия исключения или сохраняется ссылка на оригинал)

Сохранение исключения в объекте future. Пример:

```
double square_root(double x) {
if (x<0) { throw std::out_of_range("x<0"); }
return sqrt(x);
}
int main(){
   std::future<double> f = std::async(square_root, -1);
   double y=0;
   try {
         y = f.get();
    }
   catch (std::out_of_range& e){ std::cout << e.what(); }
//или catch (std::exception& e){ std::cout << e.what(); }
}
```