«Сделай Сам»:
Пул потоков своими руками

Денис Кормалев

Москва*, 2020

Альтернативы

- Boost.Asio
  - Хорошая производительность, но это Boost
- Qt
  - Большая зависимость и не самая лучшая производительность
- Folly
  - Хорошее API, много возможностей, но зависимость от Folly
- Intel TBB
  - Хорошая производительность, но не самое лучшее API; очень много бойлерплейта, если надо
- threadpoolcpp (by inkooboo)
  - Производительность с которой невозможно спорить, но полное отсутствие функционала сверх базового
- Прочие

А если бы мы хотели использовать futures?

- std::future
  - Часть стандартной библиотеки, но отсутствие какого-либо функционала
- boost::future
  - Есть then (больше ничего нет), но требует boost
- QFuture
  - Практически невозможно использовать за пределами QtConcurrent
- Folly
  - Отличное API, богатый функционал, но часть Folly
- Прочие

А если и futures и пул потоков?

- Boost.Asio
- Folly
- Прочие

Может напишем свое решение?

Futures

- C++17 как единственная зависимость
- Обработка ошибок через алгебраические типы (Either-подобное)
- Преобразования
  - T -> U, T -> Future<U>
  - Failure -> T, Failure -> Future<T>, Failure1 -> Failure2
  - …
- Дополнительный сахар
  - Future<T>, Future<U> -> Future<tuple<T, U>>
  - Container<Future<T>> -> Future<Container<T>>
  - …

Планирование задач

- C++17 как единственная зависимость
- Краткое, но, при этом, гибкое API
- Как можно меньше бойлерплейта на стороне юзера
- Использование futures
- Внутренние пулы
- Приоритизация задач
- Сахар
  - Одна функция – много входных данных
  - …

Этот доклад о

- Пуле потоков и основной реализации планировщика
- Улучшении кода шаг за шагом
- Бенчмарках
- Небольших оптимизациях, приводящих к серьезной разнице в производительности
- Дополнительных хелперах для облегчения жизни

В этом докладе НЕТ

- Реализации Futures
- “parallel_for” хелперов
- Noexcept и спецификаторов доступа
- Геттеров, сеттеров и прочего
- Дополнительных оптимизаций и подстроек, основанных на предположениях
- Все это можно найти в исходниках на гитхабе

- One more thing. Этот доклад не про «самый быстрый в мире» планировщик

Futures API

- Класс Future<T, FailureT>
  - f.fillSuccess(someVal)
  - f.fillFailure(someFailure)
  - f.onSuccess(func)
  - f.onFailure(func)
  - f.map(func)
  - f.flatMap(func)
- Класс Promise<T, FailureT>
  - promise.success(someVal)
  - promise.failure(someFailure)
  - promise.future()

Первая наивная попытка

- Очередь задач
- Вектор потоков
- Дополнительный поток для распределения задач

```
struct Worker {
  TaskInfo task;
  std::mutex mainLock;
  std::conditional_variable waiter;

  bool setTask(TaskInfo& x) {
    std::unique_lock lock(mainLock);
    if (task) return false;
    task = std::move(x);
    waiter.notify_one();
    return true;
  }

  void run() {
    while (true) {
      TaskInfo taskCopy;
      {
        std::unique_lock lock(mainLock);
        std::swap(taskCopy, task);
      }
      if (taskCopy) taskCopy.task();
      std::unique_lock lock(mainLock);
      if (!task) waiter.wait(lock);
    }
  }
};
```

```
class TaskDispatcher {
  std::vector<Worker*> workers;
  std::deque<TaskInfo> tasksQueue;
  std::mutex mainLock;

  void maintenance() {
    while(true) {
      std::this_thread::yield();
      std::unique_lock lock(mainLock);
      bool found = true;
      while (found && !tasksQueue.empty()) {
        found = false;
        for (Worker* w : workers) {
          if (w->setTask(tasksQueue.front())) {
            tasksQueue.pop_front();
            found = true;
            break;
  }}}}}

public:
  void insertTaskInfo(TaskInfo&& task) {
    std::unique_lock lock(mainLock);
    tasksQueue.push_back(std::move(task));
  }
};
```

Нам правда это нужно?

```
void maintenance() {
  while(true) {
    std::this_thread::yield();
    std::unique_lock lock(mainLock);
    bool found = true;
    while (found && !tasksQueue.empty()) {
      found = false;
      for (Worker* w : workers) {
        if (w->setTask(tasksQueue.front())) {
          tasksQueue.pop_front();
          found = true;
          break;
}}}}}
```

Отказываемся от служебного потока

- Каждое добавление задачи вызывает schedule()
- Когда задача завершена – поток тоже вызывает schedule()

```
class TaskDispatcher {
  void insertTaskInfo(TaskInfo&& task) {
    {
      std::unique_lock lock(mainLock);
      tasksQueue.push_back(std::move(task));
    }
    schedule();
  }

  void taskFinished() { schedule(); }

  void schedule() {
    std::unique_lock lock(mainLock);
    if (tasksQueue.empty()) return;
    for (Worker *w : workers) {
      if (w->setTask(tasksQueue.front())) {
        found = true;
        tasksQueue.pop_front();
        break;
      }
    }
  }
};
```

Выглядит так себе

Было бы неплохо помнить какие потоки свободные в каждый момент времени

```
for (Worker *w : workers) {
  if (w->setTask(tasksQueue.front())) {
    found = true;
    tasksQueue.pop_front();
    break;
  }
}
```

```
class TaskDispatcher {
  std::unordered_set<size_t> availableWorkers;

  void taskFinished(size_t workerId) {
    {
      std::unique_lock lock(mainLock);
      availableWorkers.insert(workerId);
    }
    schedule();
  }

  void schedule() {
    std::unique_lock lock(mainLock);
    if (tasksQueue.empty() || availableWorkers.empty()) return;
    workers[*availableWorkers.begin()]->setTask(std::move(tasksQueue.front()));
    tasksQueue.pop_front();
    availableWorkers.erase(availableWorkers.begin());
  }
};
```

Этот вызов может потенциально тоже ожидать лок

```
void schedule() {
  std::unique_lock lock(mainLock);
  if (tasksQueue.empty() || availableWorkers.empty()) return;
  workers[*availableWorkers.begin()]->setTask(std::move(tasksQueue.front()));
  tasksQueue.pop_front();
  availableWorkers.erase(availableWorkers.begin());
}
```

```
void schedule() {
  std::unique_lock lock(mainLock);
  if (tasksQueue.empty() || availableWorkers.empty())
    return;

  size_t workerId = *availableWorkers.begin();
  availableWorkers.erase(availableWorkers.begin());

  auto task = std::move(tasksQueue.front());
  tasksQueue.pop_front();

  lock.unlock();
  workers[workerId]->setTask(std::move(task));
}
```

Внутренние пулы

- Два основных типа задач:
  - Вычисления
  - Ввод/вывод
- Пользователь должен иметь возможность выставлять свои лимиты для внутренних пулов
- Пользователь должен иметь возможность привязывать внутренний пул к одному потоку

```cpp
enum class TaskType : uint8_t {
  Custom = 0,
  Intensive = 1,
  ThreadBound = 2
};

struct TaskInfo {
    //...
  std::function<void()> task;
  int32_t tag = 0;
  TaskType type = TaskType::Intensive;
    //...
};

uint64_t packPoolInfo(TaskType type, int32_t tag) {
  return (type << 32) | std::max(0, tag);
}
```

```
class TaskDispatcher {
  //...
  std::list<TaskInfo> tasksQueue;
  std::map<uint64_t, int32_t> subPoolsUsage;
  std::map<int32_t, int32_t> customTagCapacities;
  bool canBeScheduled(const TaskInfo &task) { /* Some bookkeeping here */ }

  void schedule() {
    std::unique_lock lock(mainLock);
    if (tasksQueue.empty() || availableWorkers.empty())
      return;
    size_t workerId = *availableWorkers.begin();

    for (auto it = tasksQueue.begin(); it != tasksQueue.end(); ++it) {
      if (canBeScheduled(*it)) {
        availableWorkers.erase(workerId);
        auto task = std::move(*it);
        tasksQueue.erase(it);

        lock.unlock();
        workers[workerId]->setTask(std::move(task));
        break;
      }
    }
  }
};
```

Избыток сна – тоже плохо

- Что будет, если задача в очереди ожидает место во внутреннем пуле?
  - Поток вызывает taskFinished()
  - Новая задача отправляется случайному потоку (предположим что другому)
  - Первый поток засыпает, второй просыпается
- Возможно, нам не нужно так много спать?
  - Да!

```
void schedule(size_t workerId = -1) {
  std::unique_lock lock(mainLock);
  if (tasksQueue.empty() || availableWorkers.empty())
    return;
  if (workerId < 0 || !availableWorkers.count(workerId))
    workerId = *availableWorkers.begin();

  for (auto it = tasksQueue.begin(); it != tasksQueue.end(); ++it) {
    if (canBeScheduled(*it)) {
      availableWorkers.erase(workerId);
      auto task = std::move(*it);
      tasksQueue.erase(it);

      lock.unlock();
      workers[workerId]->setTask(std::move(task));
      break;
    }
  }
}
```

Привязка внутренних пулов к потокам

- В целом, мы можем запускать планировщик как обычно

Или…

- После первой задачи в таком пуле мы знаем какой поток нам будет нужен
- Основная очередь не нужна
- Можно распределять непосредственно из insertTaskInfo()
- Звучит несправедливо по отношению к другим?
  - Да – такие задачи теоретически могут быть выполнены чуть раньше
  - И нет – при обычном планировании они находятся в неравных условиях и могут ждать нужного потока гораздо дольше
- Для этого нам нужны очереди задач в каждом потоке

```
struct Worker {
  std::deque<TaskInfo> workerTasks;
  std::mutex mainLock;
  std::conditional_variable waiter;
  size_t id;

  void addTask(TaskInfo&& x) {
    std::unique_lock lock(mainLock);
    workerTasks.push_back(std::move(x));
    waiter.notify_one();
  }
};
```

```
void run() {
  while (true) {
    TaskInfo task;
    std::unique_lock lock(mainLock);
    if (!workerTasks.empty()) {
      task = std::move(workerTasks.front());
      workerTasks.pop_front();
    }
    lock.unlock();
    if (task) task.task();
    lock.lock();
    bool wasEmpty = workerTasks.empty();
    lock.unlock();
    dispatcher->taskFinished(id, task);
    if (wasEmpty) {
      lock.lock();
      if (workerTasks.empty())
        waiter.wait(lock);
    }
  }
}
```

Приоритизация

- Не все задачи одинаково полезны
- Технически приоритет это число (uint8_t)
- Но как нам адаптировать наше решение?
- Нам нужна очередь, из которой мы будем забирать так же с одного края, но вставка может быть в середину

```
class TasksList {
private:
  std::map<uint8_t, std::list<TaskInfo>> tasksQueue;
public:
  struct iterator {
    iterator &operator++();
    bool operator==(const iterator &other) const;
    bool operator!=(const iterator &other) const;
    TaskInfo &operator*() const;
    TaskInfo *operator->() const;
  };

  void insert(TaskInfo &&taskInfo);
  iterator erase(const iterator &it);
  bool empty() const;
  size_t size() const;

  iterator begin();
  iterator end();
};
```

Насколько хорош стандартный мьютекс для нашего решения?

- Мьютексы, безусловно, очень просты в использовании и надежны
- Но, иногда, они могут быть слишком тяжелы
- Попробуем заменить на легковесный примитив на atomic_flag
- Замена блокировки в потоке
  - Высокая конкурентность – примерно так же
  - Низкая конкурентность – на 10-20% меньше оверхеда
- Замена блокировки в TaskDispatcher
  - Высокая конкурентность – на 40-70% меньше оверхеда
  - Низкая конкурентность – на 10-20% меньше оверхеда

Закрываем приложение. Оно виснет!

- Цикл в run() бесконечный
- В std::thread нет terminate()
  - Можно вызвать ~thread(), но это требует terminate_handler
- Надо останавливать потоки вручную

```
class Worker {
  std::atomic_bool poisoned{false};

  Worker::~Worker() {
    poisonPill();
    if (myself.joinable())
      myself.join();
  }

  void Worker::poisonPill() {
    poisoned = true;
    waiter.notify_one();
  }

  void Worker::run() {
    while (!poisoned) {
              // ...
    }
  }
};
```

```
TasksDispatcher::~TasksDispatcher() {
  detail::SpinLockHolder lock(&mainLock);
  for (auto worker : d_ptr->allWorkers)
    delete worker;
  d_ptr->allWorkers.clear();
}
```

```
class Worker {
  std::atomic_bool poisoned{false};

  Worker::~Worker() {
    poisonPill();
    if (myself.joinable())
      myself.join();
  }

  void Worker::poisonPill() {
    poisoned = true;
    waiter.notify_one();
  }

  void Worker::run() {
    while (!poisoned) {
            // ...
    }
  }
};
```

```
class TaskDispatcher {
  std::atomic_bool poisoningStarted{false};

  TasksDispatcher::~TasksDispatcher() {
    d_ptr->poisoningStarted = true;
    detail::SpinLockHolder lock(&mainLock);
    for (auto worker : d_ptr->allWorkers)
      delete worker;
    d_ptr->allWorkers.clear();
  }

  void insertTaskInfo(TaskInfo&& task) {
    {
      detail::SpinLockHolder lock(&mainLock,
                                  poisoningStarted);
      if (!lock.isLocked())
        return;
      //...
    }
    schedule();
  }
};
```

Бенчмарки!

Бенчмарки

- Timed repost (100’000 задач на каждый поток, ~0.1ms нагрузка)
  - Близко к реальной жизни – задачи, которые что-то вычисляют и потом стартуют другие задачи
- Empty repost (1’000’000 задач на каждый поток)
  - Похоже на timed repost, но без вычислений. Хорошая нагрузка на точки синхронизации
- Timed avalanche (100’000 задач, ~0.1ms нагрузка)
  - Один поток создает кучу задач с вычислениями. Тоже отчасти похоже на реальную задачу
- Empty avalanche (100’000 задач)
  - Один поток создает кучу пустых задач. Проверка точек синхронизации с фокусом на чтение

Время в миллисекундах	Empty avalanche	Empty repost x1	Empty repost x2	Empty repost x4	Empty repost x8
Asynqro Intensive	209	4’574	4’923	8’749	16’285
Asynqro ThreadBound	226	205	374	1’046	2’694
Boost.Asio	319	1’493	1’890	1’875	2’167
Intel TBB	26	309	526	716	1’062
Intel TBB (spawn)	--	110	138	148	262
QtConcurrent	1’339	8’234	26’872	48’353	59’112
threadpoolcpp	5	33	33	35	56

Время в миллисекундах	Timed avalanche	Timed repost x1	Timed repost x2	Timed repost x4	Timed repost x8
Asynqro Intensive	99	445	477	953	204
Asynqro ThreadBound	13	34	41	44	106
Boost.Asio	9	179	195	216	41
Intel TBB	185	168	123	106	1’494
Intel TBB (spawn)	--	159	101	66	10’190
QtConcurrent	102	327	346	393	272
threadpoolcpp	8	10	11	12	23

Попробуем немного улучшить результаты

- std::unordered_set для свободных потоков == постоянная аллокация
  - Возьмем std::bitset вместо него
- std::conditional_variable тратит слишком много времени на просыпание
  - Добавим немного холостых оборотов перед уходом в сон

Время в миллисекундах	Empty avalanche	Empty repost x1	Empty repost x2	Empty repost x4	Empty repost x8
Asynqro Intensive Old	209	4’574	4’923	8’749	16’285
Asynqro Intensive No Idle	199	3’902	4’310	8’734	10’074
Asynqro Intensive	46	600	778	2’285	12’763
Asynqro ThreadBound	27	201	403	1’133	2’616
Boost.Asio	319	1’493	1’890	1’875	2’167
Intel TBB	26	309	526	716	1’062
QtConcurrent	1’339	8’234	26’872	48’353	59’112

Время в миллисекундах	Timed avalanche	Timed repost x1	Timed repost x2	Timed repost x4	Timed repost x8
Asynqro Intensive Old	99	445	477	953	204
Asynqro Intensive No Idle	84	393	413	914	122
Asynqro Intensive	55	237	231	190	110
Asynqro ThreadBound	8	27	40	37	78
Boost.Asio	9	179	195	216	41
Intel TBB	185	168	123	106	1’494
QtConcurrent	102	327	346	393	272

Добавление новых задач

```
template <typename Task>
void runAndForget(Task &&task, TaskType type, int32_t tag, TaskPriority priority) {
  insertTaskInfo(TaskInfo(
    [task = std::forward<Task>(task)]() {
      try {
        task();
      } catch (...) {
      }
    },
    type, tag, priority));
}
```

Мы вроде хотели использовать futures, разве нет?

```
template <typename Task>
auto run(Task &&task, TaskType type, int32_t tag, TaskPriority priority) {
  Promise<bool, std::string> promise{};
  std::function<void()> f = [promise, task = std::forward<Task>(task)]() {
    try {
      task();
      promise.success(true);
    } catch (const std::exception &e) {
      promise.failure(detail::exceptionFailure<std::string>(e));
    } catch (...) {
      promise.failure(detail::exceptionFailure<std::string>());
    }
  };
  insertTaskInfo(TaskInfo(std::move(f), type, tag, priority));
  return promise.future();
}
```

Уже лучше, но как получить результат отличный от Boolean?

Что может быть результатом задачи?

- Ничего (void)
  - Нам нужно уведомить о самом факте завершения, но, так как имплементация не позволяет Future<void>, будем использовать Future<bool>
- Значение типа T
  - Future<T>
- Future<T>
  - Future<T> (не Future<Future<T>>)

```
using RawResult = typename std::invoke_result_t<Task>;
using NonVoidResult = detail::ValueTypeIfFuture_T<RawResult>;
using FinalResult = std::conditional_t<std::is_same_v<RawResult, void>,
                                       bool,
                                       NonVoidResult>;

Promise<FinalResult, std::string> promise{};

std::function<void()> f = [promise, task = std::forward<Task>(task)]() {
  try {
    if constexpr (detail::IsSpecialization_V<RawResult, Future>) {
      task().onSuccess([promise](const auto &result) { promise.success(result); })
            .onFailure([promise](const auto &failure) { promise.failure(failure); });
    } else if constexpr (std::is_same_v<RawResult, void>) {
      task();
      promise.success(true);
    } else {
      promise.success(task());
    }
  }  /* catch ... */
};
//...
```

Что насчет других типов ошибок?

- Еще один тип-параметр в run()
  - Плюсы – легко в реализации
  - Минусы – не масштабируется на доблавение других фич
- Дополнительная структура-враппер
  - Минусы – не самое ожидаемое решение
  - Плюсы – позволяет добавлять другие фичи; добавляет инверсию контроля
- Враппер – наш путь

```cpp
struct RunnerInfo {
  type PlainFailure;
  //...
};

template <typename RunnerInfo>
struct TaskRunner {
  using Info = RunnerInfo;

  template <typename Task>
  static auto run(Task &&task, TaskType type, int32_t tag, TaskPriority priority) {
    using RawResult = typename std::invoke_result_t<Task>;
    using NonVoidResult = detail::ValueTypeIfFuture_T<RawResult>;
    using FinalResult = std::conditional_t<std::is_same_v<RawResult, void>,
                                           bool,
                                           NonVoidResult>;
    using FinalFailure = typename RunnerInfo::PlainFailure;
    Promise<FinalResult, FinalFailure> promise{};
    //...
  }
}
```

```cpp
struct DefaultRunnerInfo {
  using PlainFailure = std::string;
  //...
};
using DefaultRunner = TaskRunner<DefaultRunnerInfo>;

template <typename Runner = DefaultRunner, typename Task>
auto run(Task &&task, TaskType type, int32_t tag, TaskPriority priority) {
  return Runner::run(std::forward<Task>(task), type, tag, priority);
}


run(myAwesomeTask);

run<TaskRunner<MyExtraComplicatedRunnerInfo>>(myAwesomeTask);

run<MyCustomTaskRunner>(myAwesomeTask);
```

Тип ошибки все еще недостаточно обобщен

A.k.a. те самые дополнительные фичи

Преобразование ошибок

- Есть TaskRunner с my_awesome_app::Failure в качестве PlainFailure
- Есть функция, возвращающая Future<T, std::exception_ptr>, которую мы хотим запустить как задачу
- Решения:
  - Преобразующий конструктор в my_awesome_app::Failure
  - Преобразование ошибки в каждом вызове такой задачи
  - Добавление информации о преобразовании в наш RunnerInfo

```
struct RunnerInfo {
  type PlainFailure;
  constexpr static bool deferredFailureShouldBeConverted;
  template <typename DeferredFailure>
  static PlainFailure toPlainFailure(const DeferredFailure &deferred);
};

static auto run(Task &&task, TaskType type, int32_t tag, TaskPriority priority) {
  //...
  using FinalFailure =
    std::conditional_t<RunnerInfo::deferredFailureShouldBeConverted,
                       typename RunnerInfo::PlainFailure,
                       detail::FailureTypeIfFuture_T<RawResult,
                                                     typename RunnerInfo::PlainFailure>>;
  //...
  if constexpr (detail::IsSpecialization_V<RawResult, Future>) {
    auto innerFuture = task().onSuccess(/* ... */);
    if constexpr (RunnerInfo::deferredFailureShouldBeConverted) {
      innerFuture.onFailure([promise](const auto &f) {
        promise.failure(RunnerInfo::toPlainFailure(f));
      });
    } else {
      innerFuture.onFailure([promise](const FinalFailure &f) { promise.failure(f); });
    }
  }
  //...
}
```

Использование пула потоков в качестве трамплина для разрыва стека вызовов

```
Future<int, std::string> deepRecursion(int step, int limit) {
  auto result = run([step] { /* ... */ return step; });
  if (step >= limit)
    return result;
  return result.flatMap([limit](int x) -> Future<int, std::string> {
    auto f = deepRecursion(x + 1, limit);

    return f;
  });
}
```

```
Future<int, std::string> deepRecursion(int step, int limit) {
  auto result = run([step] { /* ... */ return step; });
  if (step >= limit)
    return result;
  return result.flatMap([limit](int x) -> Future<int, std::string> {
    auto f = deepRecursion(x + 1, limit);
    if (x % 1000) return f;
    return Trampoline(f);
  });
}


template <typename T, typename FailureT>
struct Trampoline {
  Future<T, FailureT> m_future;
  explicit Trampoline(Future<T, FailureT> f = Future<T, FailureT>()) : m_future(std::move(f)) {}
  operator Future<T, FailureT>() {
    Future<T, FailureT> result = Future<T, FailureT>::create();
    m_future.onSuccess([result](const T &v) {
        runAndForget([result, v] { result.fillSuccess(v); });
      }).onFailure([result](const FailureT &f) {
        runAndForget([result, f] { result.fillFailure(f); });
      });
    return result;
  }
};
```

Вопросы?

https://github.com/dkormalev/asynqro