Санкт–Петербургский государственный университет

Филипанова Мария Александровна

Выпускная квалификационная работа

Реализация внесения неисправностей для тестирования многопоточного кода на C++

Уровень образования: бакалавриат
Направление 01.03.02 «Прикладная математика и информатика»
Основная образовательная программа СВ.5005.2018 «Прикладная математика, фундаментальная информатика и программирование»
Профиль «Современное программирование»

Научный руководитель:
профессор факультета математики и компьютерных наук СПбГУ, д.ф.-м.н. Куликов Александр Сергеевич

Рецензент:
руководитель группы разработки ArangoDB GmbH, Абрамов Андрей Викторович

Санкт-Петербург
2022 г.

Содержание

Введение

Тестирование многопоточного кода является очень сложной задачей. Сложность заключается в большом количестве исполнений, которые можно наблюдать. Это является следствием того что планировщики в современных операционных системах, как правило, реализуют вытесняющую многозадачность [16] и от этого порядок исполнения становится намного более недетерминированным, а ошибки в коде, связанные с многопоточным кодом, проявляются не на всех исполнениях, а только на части из них. Такие исполнения можно охарактеризовать тем что они содержат в себе чередования между потоками, которые были не предусмотренны программистом. Проблема заключается в том, что на то, какое у нас получится исполнение, сильно влияет состояние системы, в частности нагрузка на нее и время работы и на модульных тестах поведение планировщика получается детерминированным и повторение одного и того же теста дает одни и те же исполнения [1]. Поэтому качественно увеличить число исполнений, которые будут покрыты тестами достаточно сложно.

Один из способов покрытия большего числа исполнений тестами – это стресс тестирование. Однако, чтобы создать нужные условия, может понадобиться запускать тесты, которые будут исполняться несколько часов и их падения будут непостоянными и долго воспроизводимыми, что будет очень затруднять отладку [1]. В этом случае вероятность, с которой мы будем наблюдать ошибки, примерно равна той, что будет получаться в реальном мире.

Внесение неисправностей позволяет увеличить вероятность получения исполнения, приводящего к ошибке, вставляя сон, переход на другие ядра или потоки в некоторые точки в программе, моделируя таким образом неоптимальное поведение планировщика [5]. Оно так же позволяет моделировать ситуации, которые не произошли бы в обычных условиях, например ложные падения `compare_exchange_weak` [18]. В ситуации, когда исполнение, получаемое под внесением неисправностей, качественно зависит от параметра случайности, мы можем с более быстрой скоростью исследовать пространство исполнений, где что-то может пойти не так, чем в обычных условиях.

На данный момент нами не было найдено открытого решения, кото-

рое реализует внесение неисправностей планировщика, является удобным к интегрированию в проекты не с самого начала их разработки, и к тому же поддерживает ограниченную детерминированность исполнения. Поэтому мы решили реализовать модуль для открытой библиотеки YACLib [25], который будет реализовывать внесение неисправностей и будет иметь интерфейс, позволяющий постепенно и без больших изменений в коде внедрять внесение неисправности в уже существующие проекты.

Убрать недетерминизм, привносимый много поточностью, сохранив настоящее многоядерное исполнение, невозможно, так как останется случайность, привносимая аппаратным обеспечением, но в качестве решения этой проблемы можно заменить всю многопоточность на строго программную, используя свои аналоги потоков, написанные на уровне пространства пользователя [6], которые мы далее будем называть легкими потоками. Такое решение, что не маловажно, даст нам полный контроль над получаемым исполнением, так как реализация планировщика так же выбирается нами. Однако этот вариант более сложен для внедрения целиком в большие существующие проекты и он больше всего актуален для внедрения в отдельные подмодули или в модульные тесты библиотек.

Постановка задачи

Целью данной работы является разработка модуля открытой библиотеки YACLib для тестирования многопоточного кода на C++ с использованием внесения неисправностей, предоставляющего возможность ограниченной воспроизводимости, который было бы удобно внедрять в существующие тесты. Для достижения этой цели были выделены следующие подзадачи:

- Провести обзор аналогов.
- Разработать интерфейс использования.
- Реализовать внесение неисправностей, основываясь на интерфейсе использования.
- Внедрить внесение неисправностей в тесты библиотеки YACLib.

1. Обзор предметной области

Внесения неисправностей – это техника для тестирования программного обеспечения, позволяющая проверить способность работы программы в непридвиденных условиях. Можно выделить два основных подхода – внесение неисправностей в программное обеспечение и внесение неисправностей в аппаратное обеспечение[10]. Последний остается за рамками данной работы, так как он напрямую связан с работой с аппаратным обеспечением.

Важно заметить, что внесение неисправностей не генерирует нужные входные данные и тесты для программы, а эффективность его использования зависит от наличия достаточного числа модульных и нагрузочных тестов. Внесение неисправностей позволяет увеличить вероятность возникновения ошибок в программе при их исполнении и этим увеличить покрытие кода.

1.1. Возможные подходы к внесению неисправностей для многопоточных C++ программ

Основная часть внесения неисправностей как техники тестирования – это моделирование неисправностей разных сущностей, участвующих в исполнении программы. В общем случае это может быть моделирование не только неисправностей планировщика, но и неисправностей сокетов, сетевого оборудования, диска или моделирование перезагрузки каких-то частей системы [10] [22]. В рамках данной работы мы будем говорить только о моделировании неисправностей планировщика. Подходы могут отличаться тем, как именно неисправности вносятся в программу, однако конкретное решение может содержать сразу несколько из них.

1.1.1 Внесение неисправностей раз в определенное время

Этот способ подразумевает создание прерывания раз в определенное время, в котором будет вызываться внесение неисправностей. Такой способ не привязывает внесение неисправностей ни к каким конкретным событиям, и поэтому имеет очень небольшую вероятность создания условий для воспроизведения ошибок [1], так что он в основном подходит для моделирования

временных неисправностей или нерегулярных неисправностей аппаратного обеспечения. [10].

1.1.2 Внесение неисправностей при возникновении определенных событий.

Это может быть как замена настоящих функций на функции моделирующие неисправность, так и например установка обработчиков на определенные события – например на взаимодействие с мьютексом. Плюс этого подхода в том, что у нас появляется знание о том, где именно в программе мы находимся когда вносим неисправность. Частный случай этого подхода – снятие с исполнения вокруг взаимодействий потоков друг с другом – показывает себя хорошо в нахождении ошибок многопоточного кода [5].

1.1.3 Использование собственного планировщика.

Такой подход дает больше свободы для моделирования неоптимального поведения планировщика. Например, это позволяет рандомизированно выбирать потоки для исполнения, что при хорошей реализации может приводить к очень хорошим вероятносям нахождения ошибок [2]. К тому же он позволяет моделировать ложные пробуждения потоков при ожидании на условной переменной.

1.2. Проверка моделей

Помимо внесения неисправностей хочется так же упомянуть проверку моделей [15] как подход к тестированию многопоточных программ. Такой подход имеет схожие цели с внесением неисправностей с точки зрения того, что он тоже используется для эксплуатации программы на различных исполнениях в поиске тех, на которых произойдет ошибка. Это решение позволяет более систематично находить ошибки, однако оно подразумевает запуск программы на всем множестве качественно различных исполнений и поэтому нацелено в основном на проверку отдельных компонентов продукта, например реализации структур данных без блокирующего ожидания, так как это

решение плохо масштабируется на большие проекты [21]. В качестве примеров реализации данного подхода для C++ можно упомянуть CDSChecker [14], Fork [13] и Relacy [17].

1.3. Существующие имплементации

1.3.1 Flow

Flow [7] – это расширение C++, которое предоставляет новые ключевые слова и модель акторов для детерминированной симуляции распределенной системы с внедрением разных неисправностей, в том числе неисправностей сети, диска, планировщика [22]. Это решение совсем не подходит для внедрения в проекты не в самом начале разработки, так как оно использует модель многопоточности, отличную от той что есть в стандартном C++. Это решение не выделено в отдельную библиотеку и является частью FoundationDB [8].

1.3.2 Twist

Twist [24] – это библиотека для внесения неисправностей для C++ кода, которая поддерживает исполнение на однопоточных легких потоках, позволяя убрать недетерминированность от многопоточного исполнения, и на потоках операционной системы и определяет примитивы синхронизации, похожие на примитивы стандартной библиотеки, через которые внедряются неисправности. Она не очень подходит для внедрения в уже существующие проекты: интерфейс примитивов для внесения неисправностей является неполным относительно примитивов стандартной библиотеки и содержит лишь часть нужного интерфейса. К тому же эта библиотека подразумевает использование ее тестирующего фреймворка, из-за чего придется вносить значительные изменения в тесты.

1.3.3 ClickHouse ThreadFuzzer

ClickHouse ThreadFuzzer [3] – это решение, которое сочетает в себе первые два подхода внесения неисправностей из описанных – оно использует внесение неисправностей раз в определенное время через сигналы, а

так же регистрирует обработчики на события связанные с использованием `pthread_mutex`. Это решение не выделено в отдельную библиотеку и является частью большого репозитория.

1.3.4 Сравнение аналогов

В качестве критериев для сравнения будем рассматривать следующие:

- Возможность внедрения в существующие проекты
- Доступно в качестве библиотеки
- Многопоточное исполнение
- Воспроизводимость исполнения

Первые три описывают возможность и удобство внедрения в существующие проекты, а воспроизводимость исполнения является важным для удобства отладки тестов. Возможность внедрения в существующие проекты описывает теоритическую возможность использования решения без переписывания большей части программного кода. Доступность в качестве библиотеки является важным фактором, так как влияет на удобство использования и обновления решения. Наличие многопоточного исполнения становится важным, так как большая часть приложений не сможет запуститься и корректно работать без аппаратной многопоточности, например из-за блокирующего ожидания в зависимостях.

Таблица 1: Сравнение существующих решений

Решение	**Возможность внедрения в существующие проекты**	**Воспроизводимость исполнения**	**Доступно в качестве библиотеки**	**Многопоточное исполнение**
Flow	нет	да	нет	нет
Twist	нет	огр.	да	да
ClickHouse ThreadFuzzer	да	нет	нет	да

1.4. Выводы

В результате анализа существующих решений и предметной области в качестве точек внесения неисправностей было решено использовать точки многопоточного взаимодействия и вносить неисправности до и после работы с примитивами синхронизации, так как ошибки многопоточного программирования можно описать последовательностью нескольких конкуррентных взаимодейтсвий, из-за чего точки до и после таких взаимодейтсвий являются хорошими местами для переключений контекстов с целью выявления ошибок [1]. Так же такое решение показывает хорошие результаты на практике [5] и не требует больших преобразований в исходном коде.

Так же было решено поддержать для пользователя возможность выбора из 2 контекстов исполнений – исполнение на потоках операционной системы и исполнения на легких потоках, планируемых на 1 поток операционной системы.

Первое решение допускает постепенное внедрение в приложение и может использоваться совместно с санитайзерами потоков. Второе решение позволяет получать воспроизводимые исполнения, но требует замены объявления создания примитивов синхронизации и потоков на создание предоставленных нами аналогов во всем исходном коде и зависимостях. Воспроизводимость можно будет получить не во всех вариантах программ – например в случае взаимодействия с сетью появится новый источник случайности, который сделает исполнения менее воспроизводимыми, однако он все равно имеет меньший вклад в неудобство отладки, чем недетерминированность от многопоточного исполнения.

2. Архитектура и особенности реализации

2.1. Заголовочные файлы yaclib_std и обертки для внесения неисправностей

Так как было решено в качестве точек внесения неисправностей использовать точки многопоточного взаимодействия, нам потребуется реализовать замену для примитивов синхронизации стандартной библиотеки и заменить использования станартных примитивов на использование наших примитивов.

Можно выделить два возможных способа это сделать – динамически без преобразования кода, например с использованием механизма `LD_PRELOAD` или статически путем замены их в коде. Мы решили использовать второй вариант, в первую очередь потому что он позволяет совмещать обычные примитивы из стандартной библиотеки и примитивы с внесением неисправностей, когда мы хотим вносить неисправности только в часть кода. К тому же, у первого решения есть ряд минусов: то, какие именно функции нам нужно переопределять зависит от системы и компилятора [4], поэтому нужно делать разные реализации для разных систем и компиляторов, а так же есть ряд ограничений связанных с тем, что его нельзя использовать на программах, которые запущены от лица других пользователей или групп [11].

Заголовочные файлы с примитивами, предназначенные для использования пользователя, полностью повторяют по наполнению заголовочные файлы стандартной библиотеки, чтобы упростить необходимые преобразования кодовой базы. Эти файлы подключают другие заголовочные файлы, каждый из которых в свою очередь содержит в себе конктерный примитив (Рис. 1).

В заголовочных файлах из `yaclib_std` происходит только выбор реализации примитивов на основе значения макроса, которое определяется на этапе сборки в зависимости от того, какие агрументы были переданны. Примитивы вводятся с помощью ключевого слова `using`, и поэтому в случае исполнения без внесения неисправностей с использованием настоящих примитивов стандартной библиотеки не будет накладных расходов.

У каждого примитива есть шаблонный класс, который принимает имплементацию примитива в качестве шаблонного параментра и оборачивает

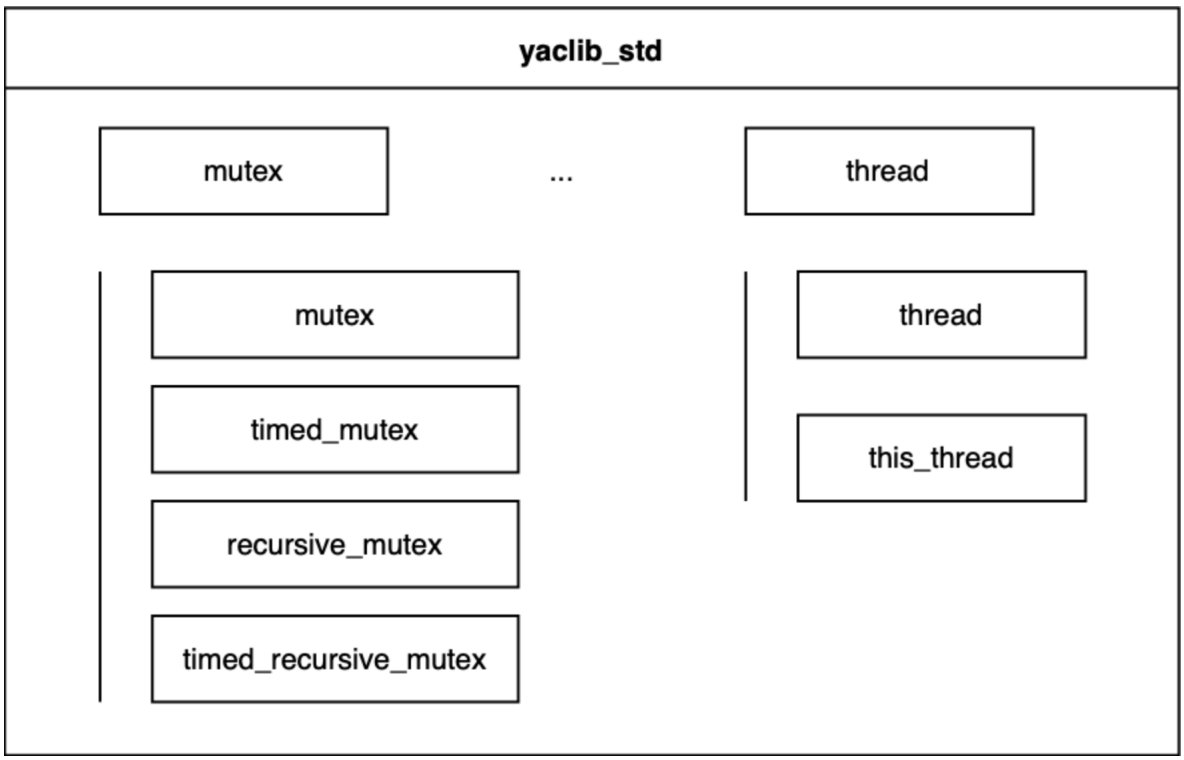

Рис. 1: yaclib_std.

вызовы его методов внесением неисправностей, а так же наследуется от нее чтобы переиспользовать конструкторы. Это позволяет избежать дублирования кода.

Для внесения неисправностей в примитивы эти классы обращаются к классу Injector, который описан в главе 2.5.

2.2. Интерфейс использования

Так как основная цель – это простота использования, нам важно, чтобы для внедрения нашего решения требовались лишь небольшие преобразования исходного кода и тестов.

Основное изменение – замены примитивов из стандартной библиотеки на классы с совершенно аналогичным интерфейсом – нужно будет только заменить подключение заголовочных файлов стандартной библиотеки на наши и добавить пространство имен к типам переменных или вызовам функций из стандартной библиотеки:

```
...
#include <yaclib_std/atomic>
#include <yaclib_std/thread>
...

void RacyCounter() {
  yaclib_std::atomic_size_t counter{0};

  static const std::size_t kIncrements = 5;
  yaclib_std::thread t1([&] {
    for (std::size_t i = 0; i < kIncrements; ++i) {
      auto old = counter.load(std::memory_order_relaxed);
      counter.store(old + 1, std::memory_order_relaxed);
    }
  });

  yaclib_std::thread t2([&] {
    for (std::size_t i = 0; i < kIncrements; ++i) {
      counter.fetch_add(1, std::memory_order_relaxed);
    }
  });
  t1.join();
  t2.join();
  ASSERT_TRUE(counter.load() == 2 * kIncrements);
}
```

Этот тест при исполнении с внесением неисправностей на основе потоков операционной системы падает в зависимости от параметров с менее чем 50 итерациями, хотя без внесения неисправностей ему для падения в большинстве случаев не хватает и 10000 итераций.

Такой подход частично покрывает и пользовательсткие примитивы синхронизации – в реализации многих из них используются какие-то из стандартных примитивов синхронизации, например атомики, но при желании

пользователь сможет добавить в их код внесение неисправностей через метод `InjectFault`, предоставляемый библиотекой.

Так же нужно будет заменить объявление `thread_local` переменных, вот пример для `static thread_local IExecutor*`:

```
static YACLIB_THREAD_LOCAL_PTR(IExecutor)
        tlCurrentThreadPool{&MakeInline()};
```

Помимо изменений в самой кодовой базе, при желании запускать тесты с внесением неисправностей на основе легких потоков, необходимо будет изменить запуск тестов:

```
#if YACLIB_FAULT == 2
  // запуск тестов с использованием легких потоков
  ::testing::UnitTest::GetInstance()->listeners()
    .Append(new test::MyTestListener());
  yaclib::fault::Scheduler scheduler;
  yaclib::fault::Scheduler::Set(&scheduler);
  // запуск легкого потока - родителя
  yaclib_std::thread tests([&]() {
    result = RUN_ALL_TESTS();
  });
  tests.join();
  YACLIB_ERROR(scheduler.IsRunning(),
    "scheduler is still running when tests are finished");
#else
  // запуск тестов без использования легких потоков
  result = RUN_ALL_TESTS();
#endif
```

Если пользователь будет использовать только внесение неисправностей на основе потоков операционной системы, то допускается замена только части примитивов на примитивы из `yaclib_std`.

2.3. Конфигурация получаемого исполнения

В нашем решении есть несколько способов по-разному влиять на исполнение.

Основной – это выбор сценария исполнения, он просходит через аргумент сборки
`YACLIB_FAULT`, возможные значения которого – `OFF`, `THREAD`, `FIBER`.

Помимо него, есть функций, позволяющие конфигурировать отдельные части исполнения:

- `SetFaultFrequency` позволяет задать примерную частоту внесения неисправносте вызовом `InjectFault`.
- `SetSeed` позволяет задать параметр случайности используемый для внесения неисправностей и планирования.
- `SetFaultSleepTime` позволяет задать максимальное время сна при использовании сна для снятие потока с исполнения в методе `InjectFault`.
- `SetAtomicFailFrequency` позволяет задать частоту ложного возвращения `false` методом `compare_exchange_weak`.
- Несколько функций, конфигурирующие исполнение на легких потоках, например размер стека, длинна одной итерации планирования и др.

2.4. Моделирование редких событий

Используя внесение неисправностей, мы так же можем моделировать события, которые обычно никогда не происходят при исполнении тестов. В нашем решении мы моделируем ряд таких событий:

- Ложные падения `compare_exchange_weak`.
  При вызове оберток этих методов, вызывается `ShouldFailAtomicWeak`, который возвращает `true` если `compare_exchange_weak` должен вернуть false, вне зависимости от того, должен ли он это делать семантически [18], что является допустимым спецификацией поведением.

Частота, с которой `ShouldFailAtomicWeak()` будет возвращать `true`, может быть сконфигурирована пользователем.

- Запоздалое пробуждение из ожидания с ограничением по времени.
  По спецификации [19], методы ожидания событий с органичением по времени, например ожидание на условной переменной с ограничением по времени, могут ждать заметно дольше указанного времени из-за задержек планировщика или конфликта за ресурсы. В нашем решении мы увеличиваем время ожидания таких методов на псевдослучайное число.

- Ложные пробуждения на условных переменных.
  Моделирование этого события доступно только при исполнении на легких потоках. Для его моделирования планировщик имеет доступ к списку ссылок на имеющиеся условные переменные и используя псевдослучайность пробуждает ожидающие потоки с определенной, задаваемой пользователем, частотой.

2.5. Класс Injector

Этот класс содержит в себе функции, используемые в обетках для внесения неисправностей. Само внесение неисправностей происходит не каждый раз, а с случайной частотой, которая сверху ограничивается заданным параментром, который можно конфигурировать:

```
void Injector::MaybeInject() {
  if (NeedInject()) {
#if YACLIB_FAULT == 2
    // внесение неисправностей на основе легких потоков
    yaclib_std::this_thread::yield();
#else
    // внесение неисправностей на основе потоков
    // операционной системы
    yaclib_std::this_thread::sleep_for(
        std::chrono::nanoseconds(1 + GetRandNumber(sleep_time)));
```

```
#endif
  }
}

bool Injector::NeedInject() {
  if (_pause) {
    return false;
  }
  if (++_count >= yield_frequency) {
    Reset();
    return true;
  }
  return false;
}

void Injector::Reset() {
  _count = GetRandNumber(yield_frequency);
}
```

Такое решение позволяет регулировать вклад внесения неисправностей в увеличение времени исполнения, а конфигурируемая случайность позволяет исследовать пространство исполнений меняя параметр случайности.

Так как `std::this_thread::yield` не обязательно снимает поток с исполнения, например в операционной системе Linux yield не будет иметь эффекта если у планировщика нет кандидатов с таким же приоритетом или выше [20], мы будем вызывать `std::this_thread::sleep_for`, чтобы гарантировать снятие с исполнения.

2.6. Реализация легких потоков для Linux

Мы будем реализовывать легкие потоки, которые будут кооперативно исполняться на одном потоке операционной системы и иметь собственные стеки. Плюс такого решения – избавление от недетерминированности, привносимой многопоточностью.

Основные компоненты тут это реализация переключения контекста, планировщика и нужных сущностей стандартной библиотеки, связанных с взаимодействием между потоками, временем и генератором случайных чисел.

2.6.1 Планировщик

При запуске первого легкого потока запускается цикл планирования, который выполняется пока есть готовые к исполнению или спящие легкие потоки:

```
void Scheduler::RunLoop() {
  while (!_queue.empty() || !_sleep_list.empty()) {
    if (_queue.empty()) {
      // отмотать время вперед до легкого потока с
      // минимальным оставшимся временем сна
      AdvanceTime();
    }
    // разбудить тех, у кого вышло время сна
    WakeUpNeeded();
    YACLIB_INFO(_queue.empty(), "Potentially deadlock");
    auto* next = GetNext();
    current = next;
    // увеличить время на фиксированную величину 1 шага
    // планирования
    TickTime();
    next->Resume();
    if (next->GetState() == detail::fiber::Completed &&
        !next->IsThreadlikeInstanceAlive()) {
      delete next;
    }
  }
  current = nullptr;
}
```

На каждом шаге планирования мы будим спящие легкие потоки, вре-

мя сна которых истекло, достаем следующий легкий поток из очереди и переходим к его исполнению. Вместо часов из стандартной библиотеки мы используем свое, модельное время. Оно является состоянием планировщика и увеличивается на константную величину на каждой итерации планирования. Мы используем модельное время для воспроизводимости, так как время системы будет меняться от запуска к запуску. К тому же, это позволяет нам будить потоки раньше времени, прокручивая время вперед, когда нет потоков в основной очереди. Вытеснений легких потоков не происходит – наша реализация многопоточности полностью кооперативная. При выборе следующего потока для исполнения мы будем выбирать случайный из текущей очереди:

```
detail::fiber::Fiber* Scheduler::GetNext() {
  YACLIB_DEBUG(_queue.empty(), "Queue can't be empty");
  auto* next = PollRandomElementFromList(_queue);
  return next;
}

BiNode* PollRandomElementFromList(BiList& list) {
  auto rand_pos = detail::GetRandNumber(
    fault::_random_list_pick == 0 ? list.GetSize()
        : fault::_random_list_pick * 2);
  auto* next = list.GetNth(list.GetSize() -
    fault::_random_list_pick + rand_pos);
  list.Erase(next);
  return next;
}
```

Такая стратегия позволит нам получать разные исполнения в зависимости от конфигурируемого параметра случайности.

2.6.2 Работа с контекстами

Было решено ограничить поддерживаемую операционную систему до Linux использованием `ucontext` [23] для работы с контекстами. Такое решение позволяет избежать необходимости реализации переключения контекста

для всех популярных архитектур процессора на ассемблере. К тому же, ограничение операционной системы до Linux не выглядит сильным упущением, так как в случае однопоточного исполнения легких потоков мы почти полностью изолированы от планировщика, а значит меньше зависим от особенностей операционной системы. К тому же, Linux является самой распространенной операционной системой для серверов. При желании в будущем можно будет добавить поддержку и других операционных систем.

Для управления контекстом выделен класс `ExecutionContext`, который хранит в себе экземпляр структуры `ucontext_t`.

Переключение контекста с помощью ucontext будет выглядить как вызов функции `swapcontext` в который в качестве агрументов мы передаем контекст на который нужно переключится и объект, в который нужно будет сохранить наш текущий контекст:

```
void ExecutionContext::SwitchTo(ExecutionContext& other) {
  swapcontext(&_context, &other._context);
}
```

Настройку стека для первого вызова так же можно сделать через функции ucontext'a:

```
void ExecutionContext::Setup(Allocation stack,
                             Trampoline trampoline, void* arg) {
  if (getcontext(&_context) == -1) {
    abort();
  }
  _context.uc_stack.ss_sp = stack.start;
  _context.uc_stack.ss_size = stack.size;
  makecontext(&_context,
    reinterpret_cast<void (*)()>(trampoline), 1, arg);
}
```

Статический метод Trampoline является методом, который будет запускаться в легком потоке при его создании. Он находится в классе `Fiber` – это

шаблонный класс, который наследуется от основного класса легкого потока `FiberBase`. Класс `Fiber` нужен, чтобы принять функцию, которая будет исполняться в этом легком потоке и ее аргументы, поэтому параметрами шаблона являются типы функции и агрументов. Он отделен от основного класса `FiberBase`, чтобы делать шаблонными только те функции, которым нужно работать с состоянием запускаемой функции и ее агрументов.

2.6.3 Аналоги примитивов стандартной библиотеки

Основа для реализации мьютексов и условных переменных для легких потоков – очередь, которая позволяет ждать и оповещать ожидающих потоков. Такая очередь не требует использования мьютекса, потому что легкие потоки реализуют невытесняющую многопоточность:

```
struct NoTimeoutTag {};

class FiberQueue {
 public:
  FiberQueue() = default;
  FiberQueue(FiberQueue&& other) = default;
  FiberQueue& operator=(FiberQueue&& other) noexcept;

  bool Wait(NoTimeoutTag);

  template <typename Rep, typename Period>
  bool Wait(const std::chrono::duration<Rep, Period>& duration) {
    return Wait(duration + SteadyClock::now());
  }

  template <typename Clock, typename Duration>
  bool Wait(const std::chrono::time_point<Clock,
            Duration>& time_point) {
    auto* fiber = fault::Scheduler::Current();
    auto* queue_node = static_cast<BiNodeWaitQueue*>(fiber);
```

```
    _queue.PushBack(queue_node);
    auto* scheduler = fault::Scheduler::GetScheduler();
    scheduler->Sleep(time_point.time_since_epoch().count());
    bool res = _queue.Erase(queue_node);
    return res;
  }

  void NotifyAll();

  void NotifyOne();

  bool Empty() const;

  ~FiberQueue();

 private:
  BiList _queue;
};
```

Ожидание с ограничением по времени реализовано через сон. Мы кладем поток сразу в 2 списка – в список спящих потоков и в список ожидающих потоков очереди. После этого поток мог быть поставлен на исполнения после того как его оповестили, либо после того как время вышло. То, что поток проснется тут только 1 раз, достигается тем что мы используем двусвязные списки для сна и ожидающих потоков и убираем его из списка в котором он еще остался: в случае если это очередь ожидания примитива сразу после пробуждения в методе `Wait`, а в случае если это список спящих потоков, то после выхода из сна в методе `Sleep` планировщика.

Пример реализации метода `lock` мьютекса с помощью очереди:

```
void Mutex::lock() {
  while (_occupied) {
    _queue.Wait(NoTimeoutTag{});
  }
```

```
  _occupied = true;
}
```

Для обертки `std::thread` сделан отдельный класс, который имеет указатель на экземпляр соответствующего ему класса легкого потока. При вызове метода `join` в основном классе легкого потока `FiberBase` указывается ссылка на поток, который его ожидает:

```
void Thread::join() {
  YACLIB_ERROR(_joined_or_detached,
               "already joined or detached on join");
  _joined_or_detached = true;
  while (_impl->GetState() != Completed) {
    _impl->SetJoiningFiber(
        fault::Scheduler::GetScheduler()->Current()
    );
    fault::Scheduler::GetScheduler()->Suspend();
  }
  AfterJoinOrDetach();
}
```

Легкому потоку при завершении нужно будет поставить на исполнение поток, который его ждал.

Кроме примитивов синхронизации и времени необходимо так же перекрыть источники случайности. В C++ не псевдо случайность обычно достигается использованием времени в качестве seed или использованием `std::random_device`. Так как для времени мы создаем обертки, использующие модельное время планировщика, осталось только сделать свой `random_device`, который внутри будет использовать генератор с фиксированным параметром случайности, например `std::mt19937_64`.

3. Тестирование

Для тестирования решения было решено осуществить его внедрение в библиотеку YACLib. Помимо этого, для части из функциональности внесения неисправностей, которая не используется в библиотеке YACLib, были написаны модульные тесты с использованием GoogleTest [9], чтобы убедиться в ее корректности.

3.1. Результаты внедрения в библиотеку YACLib

В процессе внедрения мы запускали существующие тесты библиотеки YACLib с использованием нашего решения. В библиотеке всего около 300 тестов, это тесты на остальную функциональность, не включающие тесты на внесение неисправность. Без внесение неисправностей у библиотеки 100% покрытие строк кода и покрытие ветвлений около 70%.

Библиотека была протестирована с использованием обоих контекстов исполнения, тесты были успешно запущены с использованием нашего решения более 20 раз. В результате внедрения была найдена ошибка, связанная с состоянием гонки при остановке lock-free реализации потока-исполнителя. Так же были найдены ошибки в нескольких тестах, из-за которых они могли переодически не проходить.
Так же в результате внедрения увеличились показатели покрытия кода тестами, в частности покрытие функций и покрытие ветвлений:

Таблица 2: Параметры покрытия кода тестами

Тип запуска	**Покрытие ветвлений**	**Покрытие функций**
OFF	265 (71.8%)	5112 (74.9%)
OFF x3	269 (72.5%)	5234 (75.1%)
OFF + THREAD + FIBER	276 (74.0%)	5324 (75.2%)

Измерения проводились с помощью утилиты lcov [12].

Заключение

В ходе данной работы удалось реализовать внесение неисправностей, которое удобно внедрять в проекты не с самого начала их разработки, которое так же поддерживает частичную воспроизводимость исполнения, благодаря опционального исполнения на легких потоках, исполняющихся на одном потоке операционной системы.

Из этого следует, что решение имеет свойства, которых не доставало известным аналогам:

Таблица 3: Сравнение с аналогами

Решение	**Возможность внедрения в существующие проекты**	**Воспроизводимость исполнения**	**Доступно в качестве библиотеки**	**Многопоточное исполнение**
Flow	нет	да	нет	нет
Twist	нет	огр.	да	да
ClickHouse ThreadFuzzer	да	нет	нет	да
YACLib	да	огр.	да	да

Таким образом, в ходе выполнения данной бакалаврской работы были получены следующие результаты:

- Сделан обзор предметной области. Рассмотрены основные подходы к внесению неисправностей и проведен обзор известных реализаций внесения неисправностей.
- Разработан интерфейс использования, с помощью которого было бы удобно внедрять внесение неисправностей в существующие проекты.
- Реализовано внесение неисправностей на основе потоков операционной системы.
- Реализовано внесение неисправностей на основе легких потоков, исполняющихся на одном потоке операционной системы, предоставляющее возможность ограниченной воспроизводимости.

- Произведена аппробация на тестах библиотеки YACLib.

Код работы: https://github.com/YACLib/YACLib/commits/myannyax/fault-fibers-1thread

В качестве возможного пути дальшейшего развития можно рассмотреть реализацию исполнения на легких потоках, исполняющихся на нескольких потоках операционной системы. Такое решение так же будет иметь гибкую конфигурацию исполнения, как и исполнение легких потоков на одном потоке операционной системы, но в отличии от него, это решение не будет требовать избавление от всего блокирующего ожидания в программе.

Список литературы

[1] Yosi Ben-Asher и др. “Producing Scheduling That Causes Concurrent Programs to Fail”. В: *Proceedings of the 2006 Workshop on Parallel and Distributed Systems: Testing and Debugging*. PADTAD ’06. Portland, Maine, USA: Association for Computing Machinery, 2006, с. 37—40. ISBN: 1595934146. DOI: 10.1145/1147403.1147410. URL: https://doi.org/10.1145/1147403.1147410.

[2] Sebastian Burckhardt и др. “A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs”. В: *SIGARCH Comput. Archit. News* 38.1 (март 2010), с. 167—178. ISSN: 0163-5964. DOI: 10.1145/1735970.1736040. URL: https://doi.org/10.1145/1735970.1736040.

[3] *ClickHouse / ThreadFuzzer.h*. URL: https://github.com/ClickHouse/ClickHouse/blob/master/src/Common/ThreadFuzzer.h. (дата обращения 25.04.2022).

[4] *Code Injection on Linux and macOS with $LD_PRELOAD$*. URL: https://www.getambassador.io/resources/code-injection-on-linux-and-macos/. (дата обращения 25.04.2022).

[5] Y. Eytani, E. Farchi и Y. Ben-Asher. “Heuristics for finding concurrent bugs”. В: *Proceedings International Parallel and Distributed Processing Symposium*. 2003, 8 pp.-. DOI: 10.1109/IPDPS.2003.1213514.

[6] *Fiber*. URL: https://en.wikipedia.org/wiki/Fiber_(computer_science). (дата обращения 25.04.2022).

[7] *Flow*. URL: https://github.com/apple/foundationdb/tree/master/flow. (дата обращения 25.04.2022).

[8] *FoundationDB*. URL: https://www.foundationdb.org. (дата обращения 25.04.2022).

[9] *google/googletest: GoogleTest - Google Testing and Mocking Framework*. URL: https://github.com/google/googletest. (дата обращения 25.04.2022).

[10] Mei-Chen Hsueh, Timothy K Tsai и Ravishankar K Iyer. “Fault injection techniques and tools”. В: *Computer* 30.4 (1997), с. 75—82.

[11] *ld.so(8) - Linux manual page*. URL: https://man7.org/linux/man-pages/man8/ld.so.8.html. (дата обращения 25.04.2022).

[12] *linux-test-project/lcov: LCOV*. URL: https://github.com/linux-test-project/lcov. (дата обращения 25.04.2022).

[13] *Mary / fork*. URL: https://gitlab.com/mary3000/fork. (дата обращения 25.04.2022).

[14] Brian Norris и Brian Demsky. “A practical approach for model checking C/C++ 11 code”. В: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 38.3 (2016), с. 1—51.

[15] Doron A Peled, Patrizio Pelliccione и Paola Spoletini. *Model Checking*. 2008.

[16] *Preemptive multitasking*. URL: https://en.wikipedia.org/wiki/Preemption_(computing)#Preemptive_multitasking. (дата обращения 25.04.2022).

[17] *Relacy*. URL: https://github.com/dvyukov/relacy. (дата обращения 25.04.2022).

[18] *std::atomic<T>::compare_exchange_weak, std::atomic<T>::compare_exchange_strong - cppreference.com*. URL: https://en.cppreference.com/w/cpp/atomic/atomic/compare_exchange. (дата обращения 25.04.2022).

[19] *std::condition_variable::wait_for - cppreference.com*. URL: https://en.cppreference.com/w/cpp/thread/condition_variable/wait_for. (дата обращения 25.04.2022).

[20] *std::this_thread::yield - cppreference.com*. URL: https://en.cppreference.com/w/cpp/thread/yield. (дата обращения 25.04.2022).

[21] Scott D. Stoller. “Testing Concurrent Java Programs using Randomized Scheduling”. B: *Proc. Second Workshop on Runtime Verification (RV)*. T. 70(4). Electronic Notes in Theoretical Computer Science. Elsevier, июль 2002.

[22] *Testing Distributed Systems w/ Deterministic Simulation*. URL: https://www.youtube.com/watch?v=4fFDFbi3toc. (дата обращения 25.04.2022).

[23] *The GNU C Library*. URL: https://www.gnu.org/software/libc/manual/html_mono/libc.html#System-V-contexts. (дата обращения 25.04.2022).

[24] *Twist*. URL: https://gitlab.com/Lipovsky/twist/-/tree/master/. (дата обращения 25.04.2022).

[25] *YACLib*. URL: https://github.com/YACLib/YACLib. (дата обращения 25.04.2022).