Аннотация

Обеспечение и контроль качества это одни из основных задач разработчика предоставляющего программное обеспечение. Тестирование является основным способом проверки соответствия программного обеспечения предъявляемым требованиям. В работе исследуется тестирование модуля паравиртуальной сети ядра Linux. На наличие ошибок и уязвимостей тестируется самая основная функция, инициализация модуля при подключении разных виртуальных сетевых устройств. Для осуществления данной задачи был исследован один из современных подходов к тестированию программного обеспечивания ядра Linux. В качестве основного инструмента тестирования используется фреймворк KUnit встроенный в код ядра. В работе описываются как методы и подходы к тестированию так и технические детали разработки тестов, оценка качества и трудозатратности исследуемых подходов к тестированию. Представлены примеры модульных тестов, покрывающие большую часть основного функционала инициализации драйвера.

Содержание

1 Введение

В современной индустрии активно внедряются информационные технологии и в связи с этим ежегодно растёт необходимость повышения качества программного обеспечения и расширение существующего функционала. Требование безопасности и защищённости относиться к числу основных сил двигающих развитие современных компьютерных технологий. Во всём мире программное обеспечение встраивается в процессы автоматизации практически во всех сферах. Как следствие, растёт число возможных уязвимостей и ошибок, что формирует потребность в эффективной проверке и тестировании исходного кода.

Массовый потребительский сегмент получает почти неограниченный доступ к разным программам и возможностям современных технологий. Как следствие, тяжело представить современную жизнь обычного человека, если откажут финансовые, медицинские или государственные информационные экосистемы. Уязвимости и отказы программ в этих сферах могут привести к катастрофическим последствиям.

Подавляющее большинство программ обеспечивающее жизнеспособность разных индустриальных, социальных, государственных, финансовых, медицинских инфрастуктур работают на веб-серверах. Практически весь рынок веб-серверов захвачен дистрибутивами на основе ядра операционных систем Linux. Смартфоны на Android, суперкомпьютеры, космические и авиационные технологии, промышленное производство, транспорт и многие другие сферы используют ядро Linux. Как следствие, требование защищённости и безопасности программного кода ядра операционных систем Linux многократно растет с каждым годом.

В отличие от закрытых и проприетарных операционных систем, процесс разработки Linux является открытым и не заблокирован. Этот процесс является как его сильной, так и слабой стороной. Поскольку несколько раз-

работчиков продолжают добавлять новые функции и исправлять ошибки, непрерывная интеграция и тестирование жизненно важны для обеспечения того, чтобы ядро продолжало работать на существующем оборудовании по мере добавления новой аппаратной поддержки и функций. При разработке с открытым исходным кодом разработчики и пользователи разделяют ответственность за тестирование. Одновременно с этим, открытый код даёт огромную возможность для поиска уязвимостей и способов внедрения вирусных программ в разные инфрастуктуры, где используется ядро. И это еще одна важная причина тестирования исходного кода ядра.

В вышеперечисленных сферах весь налаженный процесс взаимодействия разных компонент системы осуществляется через сеть. Представить современный сервис или программу, которая не взаимодействует с сетью невозможно. Большинство угроз и атак происходят из Интернета, из-за чего самым важным становится защита интерфейсов, обеспечивающих соединение с Интернетом и с выходом в сеть.

Virtio был разработан как стандартизированный открытый интерфейс для виртуальных машин для доступа к упрощенным устройствам, таким как блочные устройства и сетевые адаптеры. Virtio-net драйвер паравиртуальной сети это виртуальная карта ethernet, которая на сегодняшний день является самым сложным устройством, поддерживаемым virtio в ядре Linux. Драйвер позволяет уменьшить накладные расходы на виртуализацию и увеличить производительность сетевого ввода/вывода. Этот драйвер является частью проекта с открытым исходным кодом KVM. Следовательно, тестировать исходный код данного драйвера является очень важной задачей для обеспечения безопасности всей инфрастуктуры.

Модульное тестирование это разновидность тестирования в программной разработке, которое заключается в проверке работоспособности отдельных функциональных модулей, процессов или частей кода программы. Оно позволяет избежать ошибок или быстро исправить их при обновлении или

дополнении ПО новыми компонентами, не тратя время на проверку программного обеспечения целиком.

1.1 Актуальность

Подведя итоги актуальность исследовательской работы можно сформулировать в следующих пунктах.

- Тестирование позволяет компаниям, занимающихся разработкой драйверов и операционных систем, улучшить свои продукты, повысить их надежность, отзывчивость, а также удовлетворить нужды своих пользователей.

- В связи с массовой потребностью операционных систем на основе ядра Linux во многих сферах индустрии, повышение безопасности и отказаустойчивости ядра приводит к повышению этих же качеств у всей системы.

- Ядро Linux постоянно развивается и совершенствуется. Тестирование драйверов паравиртуальной сети ядра Linux может помочь в оценке возможных улучшений и новых свойств, повышающих производительность и эффективность данной технологии.

- Быстрота и эффективность. Покупка новых серверов может быть очень дорогостоящей задачей. Паравиртуализация является одним из способов эффективно использовать существующий сервисный парк серверов. Тестирование драйвера паравиртуальной сети ядра Linux может подтвердить, что данная технология является быстрой и эффективной.

- Безопасность. Тестирование драйверов паравиртуальной сети ядра Linux может помочь в обнаружении уязвимостей в сетевых интерфейсах, что особенно важно для обеспечения безопасности виртуальных сред.

- Паравиртуализация под Linux с каждым годом становится все популярнее и используется во многих отраслях и сферах. Драйверы паравиртуальной сети имеют критически важное значение для успешной работы системы.

- Тестирование драйвера паравиртуальной сети ядра Linux позволяет изучить и оценить полную функциональность этого драйвера. Это позволит улучшить качество работы системы, предотвратить возможные проблемы и сбои в работе системы.

- Тестирование драйвера паравиртуальной сети ядра Linux поможет выявить и устранить проблемы совместимости драйвера с другими компонентами системы. Часто возникают проблемы при использовании сторонних драйверов, что может привести к сбоям в работе системы.

- Увеличение популярности виртуализации. Паравиртуализация становится все более распространенной технологией виртуализации, которая используется в Linux. Тестирование драйвера паравиртуальной сети ядра Linux будет актуально для оценки производительности и надежности паравиртуальных сетевых интерфейсов.

1.2 Цель и задачи

Подведя итоги целью исследовательской работы является исследование подхода к модульному тестированию применительно к коду драйверов ядра Linux на примере драйвера паравиртуальной сети virtio-net.

Для достижения поставленной цели в исследовательской работе решаются следующие задачи.

- Разработать модульные тесты для исходного кода функции инициализации драйвера паравиртуальной сети virtio-net.

- Получить оценку качества разработанных модульных тестов.

- Оценить совместимость разработанных тестов для разных версий ядра.
- Оценить трудозатратность выбранного подхода для разработки модульных тестов.
- На основе исследований сделать выводы о выбранном подходе тестирования для кода драйверов ядра Linux.

В исследовательской работе рассматривается фреймворк KUnit [1] для тестирования кода ядра Linux в частности исходного кода драйвера паравиртуальной сети virtio-net. Тесты можно запускать локально на рабочей машине разработчика, никак не ограничивая себя разными виртуальными машинами и покупкой специального аппаратного оборудования.

Также рассматриваются детали разработки модульных тестов при помощи инструмента KUnit, отладка кода тестов при помощи инструмента gdb, запуск тестов на локальной машине без привязки к разным архитектурам процессора и запуск тестов на виртуальной машине при помощи инструмента qemu для проверки работоспособности на процессорах других архитектур, как качество разработанных тестов получение покрытия для целевого тестируемого кода с использованием инструментов gcov и lcov, совместимость разработанных тестов для разных версий ядра, трудозатратность разработки модульных тестов с помощью инструмента KUnit, разные возможности, методы и практики программирования предоставленные инструментом KUnit для удобной разработки модульных тестов.

2 Тестирование программного обеспечения

2.1 Введение в тестирование

Тестирование программного обеспечения - это процесс оценки и проверки того, что программный продукт или приложение выполняет то, что

от него требуется. Это включает в себя выполнение программных или системных компонентов с использованием ручных или автоматизированных инструментов, для оценки одного или нескольких представляющих интерес свойств. Целью тестирования программного обеспечения является выявление ошибок, пробелов или отсутствующих требований в сравнении с фактическими требованиями. Преимущества тестирования включают предотвращение ошибок, снижение затрат на разработку и повышение производительности.

Выполнение тестовых действий на ранних этапах помогает сохранить усилия по тестированию на переднем плане, а не в качестве запоздалой мысли при разработке. Более ранние тесты программного обеспечения также означают, что устранение дефектов обходится дешевле.

2.2 Категории тестирования

Обычно тестирование подразделяется на три большие категории.

- Функциональное тестирование — это тип тестирования программного обеспечения, который проверяет соответствие программной системы функциональным требованиям и спецификациям. Целью функциональных тестов является тестирование каждой функции программного приложения путем предоставления соответствующих входных данных и проверки выходных данных на соответствие функциональным требованиям и спецификациям.

- Нефункциональное тестирование - это вид тестирования программного обеспечения для проверки нефункциональных параметров, таких как надежность, нагрузочный тест, производительность и подотчетность программного обеспечения. Основной целью нефункционального тестирования является проверка скорости чтения программного обеспечения в соответствии с нефункциональными параметрами.

- Регрессионное и тех обслуживающее тестирование — Выполнение тестирования после его выпуска известно как тестирование на техническое обслуживание.

Существует множество различных типов тестов программного обеспечения внутри каждой категории. Каждый из них имеет определенные цели и стратегии.

- Приемочное тестирование — проверка того, работает ли вся система должным образом.

- Интеграционное тестирование — обеспечение совместной работы программных компонентов или функций.

- Модульное тестирование — проверка того, что каждый программный модуль работает должным образом. Модуль — это самый маленький тестируемый компонент программного обеспечения.

- Функциональное тестирование — проверка функций путем эмуляции бизнес-сценариев на основе функциональных требований.

- Тестирование производительности — тестирование того, как программное обеспечение работает при различных рабочих нагрузках. Нагрузочное тестирование, например, используется для оценки производительности в реальных условиях нагрузки.

- Регрессионное тестирование — проверка того, нарушают ли новые возможности или особенности уже существующую функциональность. Тестирование работоспособности может использоваться для проверки меню, функций и команд на поверхностном уровне, когда нет времени на полный регрессионный тест.

- Нагрузочное тестирование - проверка того, какую максимальную нагрузку может выдержать система, прежде чем она выйдет из строя. На-

грузочное тестирование считается разновидностью нефункционального тестирования.

- Тестирование удобства использования - проверка того, насколько хорошо клиент может использовать систему или веб-приложение для выполнения поставленных задач.

В каждом случае проверка базовых требований является критической оценкой. Не менее важно и то, что ознакомительное тестирование помогает тестировщику или группе тестирования выявлять труднопрогнозируемые сценарии и ситуации, которые могут привести к ошибкам в программном обеспечении.

Даже простое приложение может быть подвергнуто большому количеству разнообразных тестов. План управления тестированием помогает расставить приоритеты в отношении того, какие типы тестирования обеспечивают наибольшую ценность при выделенных объемах времени и ресурсов. Эффективность тестирования оптимизируется за счет выполнения наименьшего количества тестов для выявления наибольшего количества дефектов.

2.3 Актуальность

Тестирование программного обеспечения важно, потому что, если в программном обеспечении есть какие-либо ошибки, их можно выявить на ранней стадии и устранить до поставки программного продукта. Должным образом протестированный программный продукт обеспечивает надежность, безопасность и высокую производительность, что в дальнейшем приводит к экономии времени, меньшей стоимости эффективности и удовлетворенности потребителей.

Несвоевременная доставка или дефекты программного обеспечения могут нанести ущерб репутации бренда, что приведет к разочарованию и потере клиентов. В крайних случаях ошибка или дефект могут привести к ухудше-

нию состояния взаимосвязанных систем или вызвать серьезные сбои в работе.

Хотя само тестирование стоит денег, компании могут ежегодно экономить миллионы на разработке и поддержке, если у них есть хорошая методика тестирования. Раннее тестирование программного обеспечения выявляет проблемы еще до того, как продукт выходит на рынок. Чем раньше команды разработчиков получат отзывы о тестировании, тем скорее они смогут решить такие проблемы, как

- Архитектурные недостатки
- Плохие решения проектирования
- Недопустимая или некорректная функциональность
- Уязвимости в системе безопасности
- Проблемы с масштабируемостью

Когда разработка оставляет достаточно места для тестирования, это повышает надежность программного обеспечения, и высококачественные приложения поставляются с небольшим количеством ошибок. Система, которая соответствует ожиданиям потребителей или даже превосходит их, потенциально приводит к увеличению спроса продукта.

2.4 Модульное тестирование

Модульное тестирование - это процесс разработки программного обеспечения, в ходе которого мельчайшие тестируемые части приложения, называемые модулями, индивидуально проверяются на предмет правильной работы. Основная цель модульного тестирования - изолировать написанный код для тестирования и определить, работает ли он так, как задумано.

Данный метод использует скрупулезный подход к созданию продукта посредством постоянного тестирования и доработки. Этот метод тестирования также является первым уровнем тестирования программного обеспечения, который выполняется перед другими методами, такими как интеграционное тестирование. Модульные тесты обычно изолированы, чтобы гарантировать, что модуль не полагается на какой-либо внешний код или функции.

Модульное тестирование включает в себя только те характеристики, которые жизненно важны для производительности тестируемого модуля. Это побуждает разработчиков изменять исходный код, не беспокоясь о том, как такие изменения могут повлиять на функционирование других модулей или программы в целом. Как только все модули программы заработают максимально эффективно и безошибочно, разработчики смогут оценить более крупные компоненты программы с помощью интеграционного тестирования.

Можно выполнять модульные тесты вручную или автоматически. Автоматизированные подходы обычно используют платформу тестирования для разработки тестовых примеров. Эти фреймворки также настроены на то, чтобы отмечать и сообщать о любых неудачных тестовых примерах, а также предоставлять сводку тестовых примеров. В них как правило интегрированы инструменты для обнаружения некорректных обращений к памяти, утечки памяти, интсрументы для получения покрытия целевого кода тестами, подсказки прдлагающие технические решения для найденных тестами ошибок и многое другое.

Преимущества модульного тестирования

Модульное тестирование имеет много преимуществ, в том числе следующие.

- Чем раньше выявлена проблема, тем меньше возникает сложных ошибок.

- Устранение проблем на ранней стадии обычно обходится дешевле, чем их устранение на более поздних этапах разработки. Более упрощенные

процессы отладки.

- Есть возможность сразу изминить код программы. Исправление кода занимает меньше времени.

- Появляется возможность повторного использования кода в новых проектах.

Недостатки модульного тестирования

Хотя модульное тестирование является неотъемлемой частью любой стратегии разработки и тестирования программного обеспечения, есть некоторые аспекты, о которых следует знать. К недостаткам модульного тестирования относятся следующие.

- Модульные тесты проверяют только наборы данных и их функциональность, они не будут обнаруживать ошибки при интеграции и не позволят выявить все ошибки.

- Возможно, потребуется разработать больше строк тестового кода для тестирования одной строки кода, что приведет к потенциальным затратам времени.

- Плохо работают с многопоточным кодом. Модульные тесты обычно просты, а тесты для многопоточных систем, наоборот, должны быть достаточно большими и сложными и использовать разные компоненты из разных модулей.

- Разработчикам, возможно, придется освоить новые навыки для правильного внедрения модульного тестирования например, научиться использовать определенные автоматизированные программные средства.

3 Драйвер паравиртуальной сети

Паравиртуализация это техника виртуализации в процессе работы которого гостевые операционные системы подготавливаются для исполнения внутри виртуализированной среды, путем незначительных модификаций ядер. Операционная система, которая виртуализирует гостевые системы называется хост системой или хостом. Хост система взаимодействует с гостевыми системами при помощи программы гипервизора, которая предоставляет гостевым системам интерфейс для использования разных ресурсов хост системы.

Virtio был разработан как стандартизированный открытый интерфейс для виртуальных машин, для доступа к упрощенным устройствам, таким как блочные устройства и сетевые адаптеры. Virtio-net это виртуальная карта эзернет и самое сложное устройство, поддерживаемое на сегодняшний день от virtio [2].

3.1 Фундаментальные понятия и базовые блоки

Гость это виртуальная машина, установленная, запущенная на физическом компьютере. Хост система предоставляет гостю ресурсы компьютера. У гостя есть отдельная операционная система, работающая поверх операционной системы хоста через гипервизор. Следующие базовые блоки создают среду, к которой впоследствии подключается virtio [3].

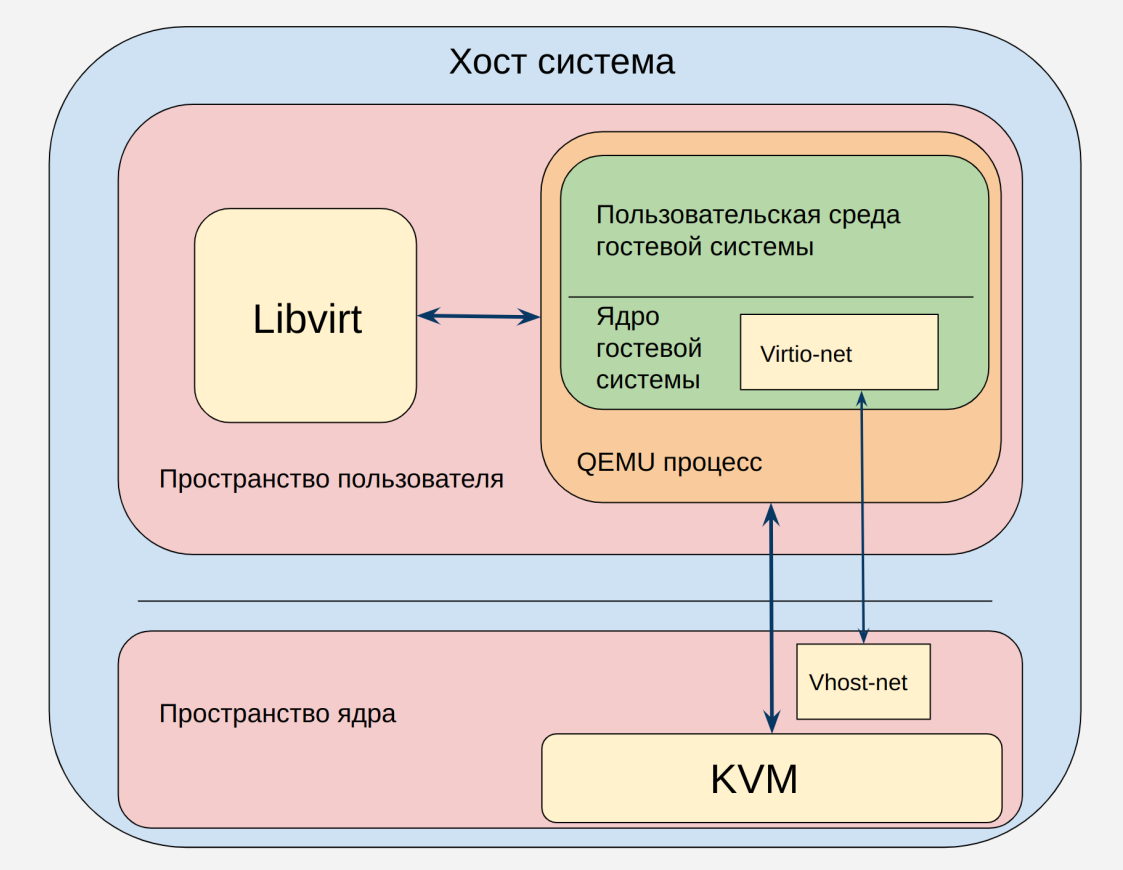

Рис. 1: Высокоуровневая архитектура

1. **KVM**

   Виртуальная машина на базе KVM ядра, которая позволяет ядру Linux функционировать в качестве гипервизора, так что хост система может запускать несколько изолированных виртуальных сред, называемых гостевыми. KVM в основном предоставляет Linux возможности гипервизора. Это означает, что компоненты гипервизора, такие, как организатор памяти, планировщик, сетевой стек и другие предоставляются как часть ядра Linux. Виртуальные машины это обычные процессы Linux, запланированные стандартным планировщиком Linux с выделенным виртуальным оборудованием, таким как сетевые адаптеры.

2. **QEMU**

   qemu это монитор для виртуальной машины, который посредством эмуляции предоставляет набор различных аппаратных средств и моделей

устройств для гостевой машины. qemu можно использовать вместе с KVM для запуска виртуальных машин с почти собственной скоростью, используя аппаратные расширения. Гостевой запуск выполняется через интерфейс командной строки qemu. Данный интерфейс предоставляет возможность указать все необходимые параметры конфигурации для qemu.

3. **Libvirt**

   Libvirt это интерфейс, который преобразует конфигурации в формате XML в вызовы командной строки qemu. Он также предоставляет демон администратора для настройки дочерних процессов, таких как qemu. Поэтому qemu не требует привилегий root. Когда Запуск виртуальной машину использует libvirt для запуска процесса qemu для каждой из виртуальных машин, вызывая по одному процессу qemu для каждой виртуальной машины.

4. **Vhost-net**

   Vhost-net протокол виртуального хостинга, который используется в ядре Linux. Протокол vhost, позволяет реализовать передачу данных, идущую непосредственно от ядра, то есть хоста, к гостевой системе, минуя процесс qemu. Это позволяет не тратить накладные расходы на каждое переключение контекста внутри гостевой системы и не обращаться каждый раз по цепочке гостевая система, процесс qemu, ядро хост системы и обратно. Однако сам протокол vhost описывает только то, как установить обмен данных. Ожидается, что тот, кто его реализует, также реализует кольцевую систему обмена данных, фактических пакетов отправки и получения. Протокол vhost может быть реализован в ядре (vhost-net) или в пользовательском пространстве (vhost-user).

5. **Virtio-net**

Virtio-net или virtio-networking это сетевое устройство virtio, стандартизированный открытый интерфейс для виртуальных машин, для доступа к упрощенным устройствам, таким как блочное хранилище и сетевые адаптеры. Это паравиртуализированное сетевое устройство, которое обладает хорошей производительностью и может быть перенесено с одной хост системы на другую. Virtio-net является реализацией протокола vhost в пространстве ядра.

4 Разработка модульных тестов

4.1 Введение

Прежде чем начать писать тесты нужно подготовить код ядра и используемые инструменты. После чего можно приступить к разработке самих тестов. Подготовку к имплементации тестов можно разбить на следующие шаги.

- Скачать выбранную для исследования версию ядра Linux.
- Установить актуальные версии компиляторов.
- Создать все нужные конфигурационные файлы для запуска тестов при помощи фреймворка KUnit.

Разработка тестов проведена на компьютере с процессором intel архитектуры x86_64 на дистрибутиве ядра операционных систем Linux Ubuntu 22.04 lts. Данный факт никак не ограничивает разработку тестов, так как фреймворк KUnit дает возможность запускать тесты для разных архитектур процессора при помощи эмулятора qemu [4]. Также есть возможность запуска тестов в режиме UserModeLinux [5].

Разработка тестов включает в себя следующие шаги.

- Добавление цели, для кода тестирования, в систему сборки ядра.

- Настройка зависимостей в конфигурационной системе ядра.
- Разработка тестов путем написания обычного программного кода.

Разработка модульных тестов для драйвера ядра является более сложной задачей, чем обычно бывает для индустриальных программ. Возникают множество разных проблем и вариантов их решения. Множество технических деталей и особенности кода приводят к непредвиденным временным затратам. Такие проблемы быстрее выявляются и решаются при помощи отладчиков и верифицирующих программ.

Аллокация и освобождение памяти являются неотъемлемой частью разработки кода ядра. Чтобы разработка тестов была более упрощённая, фреймворк KUnit дает возможность использовать специальные конструкции для более удобной работы с памятью. Так же есть возможность запустить тесты с адресным санитайзером, что упрощает процесс разработки и нахождения ошибок в коде тестов.

С целью оценки качества разработанных тестов нужно получить покрытие кода тестами. Для достижения данной цели используются инструменты gcov и lcov. Фреймворк KUnit пока не умеет работать с этими инструментами, поэтому в исследовательской работе рассматривается отдельный способ получения покрытия путем запуска тестов в эмуляторе qemu.

4.2 Подготовка

Фреймворк KUnit [1] это основной инструмент, который используется для разработки тестов и в исследовательской работе рассматриваются подходы из данного фреймворка. Данный фреймворк является частью кода ядра linux. Поэтому скачав код ядра мы сразу автоматически получим еще и инструмент KUnit. В исследовательской работе рассматриваются версии ядра **5.10.3**, **5.10.y** и **6.2.7**. Для скачивания ядра нужно воспользоваться терминалом. Находясь в директории **workspace** запускать следующую команду.

```
wget https://cdn.kernel.org/pub/linux/kernel/v5.x/linux-5.10.3.tar.xz
```

После завершения скачивания должен был появиться файл архив **linux-5.10.3.tar.xz**, разархивируем ее при помощи команды ниже.

```
tar xvf linux-5.10.3.tar.xz
```

При успешном завершении разархивации должна появиться директория **linux-5.10.3** с исходным кодом ядра.

Подавляющая часть кода ядра написана на языке **C**, поэтому нужно установить набор компиляторов. Существуют много разных версий компиляторов языка **C** для разных архитектур процессора. Чтобы в дальнейшем не возникали проблемы предлагается использовать уже собранные, готовые к работе компиляторы с официальной страницы ядра Linux [6]. Скачиваем и разархивируем следующими командами.

```
wget https://mirrors.edge.kernel.org/pub/tools/crosstool/files/bin/x86_64/11.3.0/x86_64-gcc-11.3.0-nolibc-x86_64-linux.tar.xz
tar xvf x86_64-gcc-11.3.0-nolibc-x86_64-linux.tar.xz
```

В итоге должна появиться директория **gcc-11.3.0-nolibc** в которй уже есть собранный готовый к работе компилятор. Данный набор компиляторов может быть использован только для сборок ядра. Обычные пользовательские программы невозможно будет скомпилировать.

Теперь нужно создать соответствующие файлы и конфигурации для дальнейшей работы с инструментом KUnit. Не нужно компилировать и собирать все ядро. Достаточно запустить сам фреймворк. Для этого в терминале находясь в директории **workspace/linux-5.10.3** запускаем следующую команду.

```
./tools/testing/kunit/kunit.py run
```

Для использования скаченного компилятора можно указать путь к собранному компилятору используя специальный аргумент.

```
./tools/testing/kunit/kunit.py run --cross_compile=<path>
```

После запуска команды фреймворк инициализирует стандартные конфигурации, директорию для сборки и дальнейшей работы. Без указания специального аргумента **–arch=** все конфигурации и сборку будут для UserMode Linux [5]. Чтобы указать конкретную архитектуру нужно передавать аргумент **–arch=** с соответствующим значением.

Если этапы скачивания ядра и установки компилятора прошли успешно, то после запуска команды в терминале должна быть следующая картина.

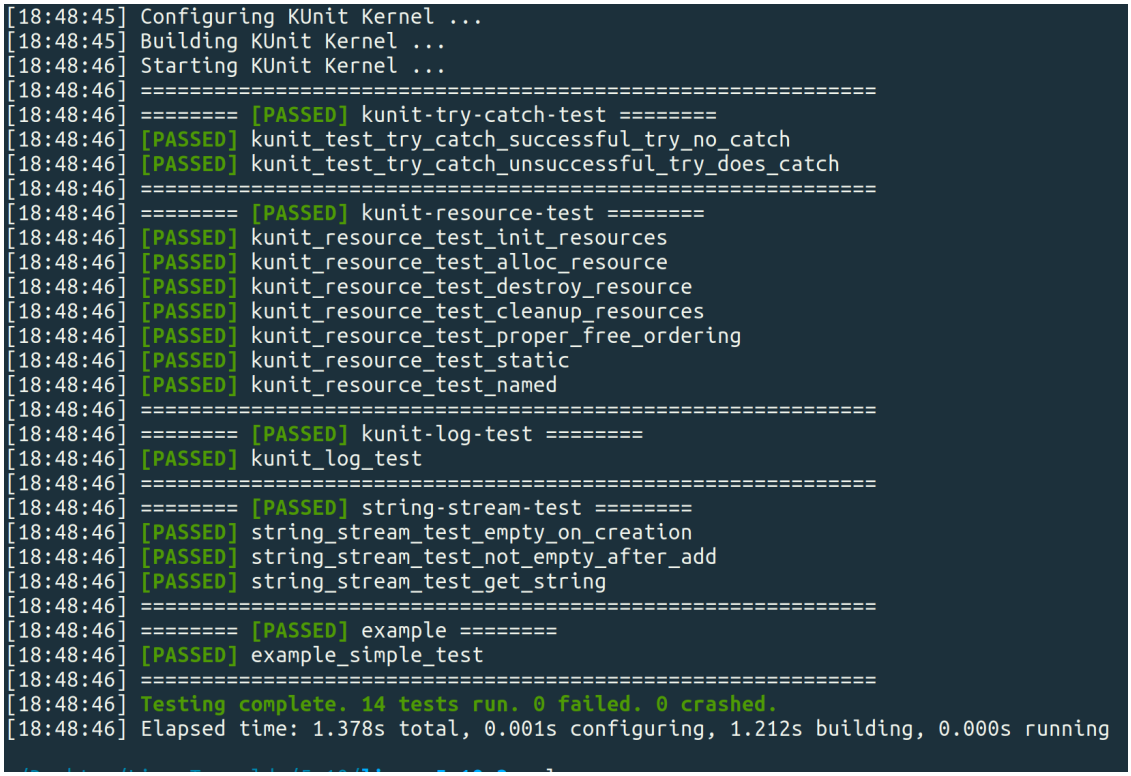

Рис. 2: Успешная работа фреймворка KUnit

На рисунке 2 видны результаты запуска некоторых модульных тестов интегрированных в данную версию ядра. Также должна появиться директория **.kunit**, где помимо разных файлов есть запускаемый бинарный файл **vmlinux**. Это и есть программа запускающая модульные тесты и при запуске создается обычный пользовательский процесс. Запуск конкретных наборов тестов указывается в кофигурационных файлах. По умолчанию запускаются те тесты, для которых опция включена.

4.3 Разработка тестов

В данном разделе будут использоваться относительные к директории linux-5.10.3 пути. Целевой код тестирования находиться в файле drivers/net/virtio_net.c. Нужно создать два файла в той же директории. Созданный файл virtio_net_test.c будет содержать код реализации модульных тестов. Созданный файл virtio_net_test.h нужен для включаемых заголовочных файлов, объявлений структур, функций, констант и макросов. Заголовочный файл virtio_net_test.h нужно включить внутри файла с кодом. Для разработки модульных тестов с использованием фреймворка KUnit нужно в заголовочный файл добавить следующий обязательный заголовок.

```
#include <kunit/test.h>
```

Структура тестов внутри файла с реализацией тестов разбивается на наборы тестов, которые содержат несколько тест-кейсов. Ниже приведен пример минимального кода модульных тестов для работы фреймворка KUnit.

```
static void example_test_case(struct kunit *test)
{}

static struct kunit_case test_cases[] = {
  KUNIT_CASE(example_test_case),
    {}
};

static struct kunit_suite test_suite = {
  .name = "virtio_net tests",
  .test_cases = test_cases,
};

kunit_test_suites(&test_suite);

MODULE_LICENSE("GPL");
```

Внутренняя структура файла с тестами

Чтобы запустить тесты одного кода недостаточно, нужно создать соответсвующую конфигурацию в системе конфигураций и добавить нужные файлы в систему сборки. Ядро Linux использует систему сборки make а для автоматической генерации путей для зависимостей используется своя систеима конфигураций KConfig [7]. Для создания конфигурации нужно открыть файл drivers/net/Kconfig и добавить следующие строки.

```
config VIRTIO_NET_TEST
    tristate "Example testing" if !KUNIT_ALL_TESTS
    depends on KUNIT=y
    default KUNIT_ALL_TESTS
```

Минимальная Конфигурация

Когда уже добавлена кофигурация, в файле drivers/net/Makefile можно указать все файлы, которые должны быть скомпилированы в цели данной конфигурации.

```
obj-$(CONFIG_VIRTIO_NET_TEST) += virtio_net_test.o
```

Добавленная цель в файле Makefile

Для запуска осталось включить созданную конфигурацию внутри файла настроек KUnit. Следующие строки внутри файла .kunit/.kunitconfig решат данную задачу.

```
CONFIG_KUNIT=y
CONFIG_VIRTIO_NET_TEST=y
```

Файл .kunitconfig

Запустив тесты привычной командой можно заметить, что в терминале появились строчки вывода подтверждающие, что был запущен новый добавленный набор тестов. Новый тест обведен в красную рамку на рисунке 3.

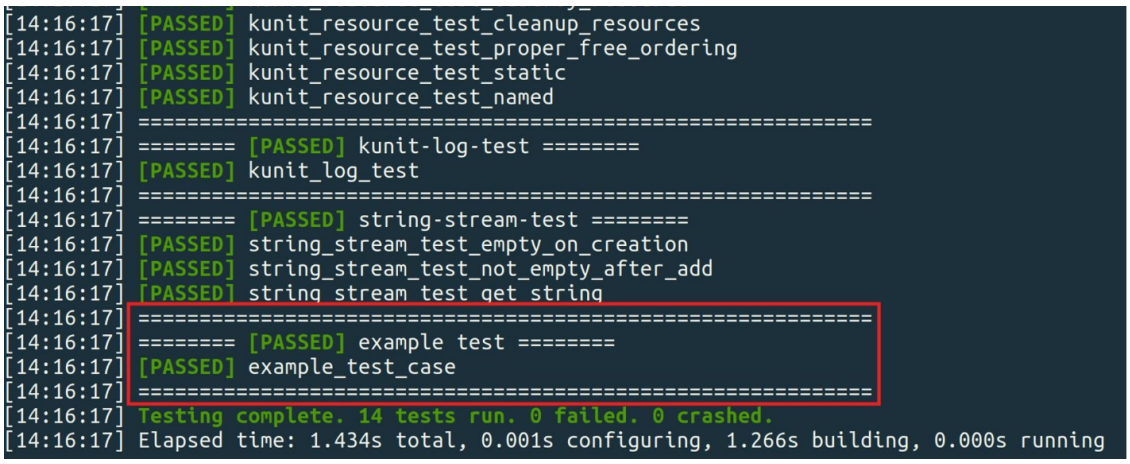

Рис. 3: Добавленный тест

Целевой код тестирования исследовательской работы это функция инициализации virtnet_probe драйвера virtio_net. Данная функция является статической функцией внутри файла virtio_net.c. В языке **C** статические функции можно вызывать только внутри текущего модуля компиляции. Это означает, что внутри созданного ранее файла для кода тестов, вызывать функцию не получиться. Чтобы решить данную проблему было принято решение включить код тестов внутрь файла virtio_net.c. Для включения тестов достаточно добавить следующие строчки в конец файла с кодом драйвера.

```
// Original virtio_net.c implementation


#ifdef CONFIG_VIRTIO_NET_TEST
#include ”virtio_net_test.c”
#endif
```

файл virtio_net.c

Данная конструкция работает следующим образом. Если внутри конфигурационного файла включена конфигурация **CONFIG_VIRTIO_NET_TEST=y**, то в конец файла драйвера добавиться код с тестами и во время компиляции тесты тоже будут включены в итоговый исполняемый файл. В остальных случаях код драйвера останется в том неизменном виде в котором был без добавления тестов.

Так как теперь тесты включены в код драйвера нужно поменять файлы для компиляции в системе сборки. Изменения затрагивают только файл **drivers/net/Makefile**. В ранее добавленной строчке меняется только объектный файл.

```
-    obj-$(CONFIG_VIRTIO_NET_TEST) += virtio_net_test.o
+    obj-$(CONFIG_VIRTIO_NET_TEST) += virtio_net.o
```

Изменения в файле Makefile

Если на данном этапе попробовать запустить тесты, то возникнет ошибка линковки. Ошибка возникает по причине того, что теперь при запуске компилируется весь код драйвера из файла virtio_net.c. Код данного файла использует разные внешние зависимости и эти зависимости нужно указать внутри конфигурации для теста. Добавление следующих строчек в файле **drivers/net/Kconfig** решает проблему.

```
    config VIRTIO_NET_TEST
     tristate ”Test for virtio_net” if !KUNIT_ALL_TESTS
     depends on KUNIT=y
+    select NETDEVICES
+    select NET_CORE
+    select VIRTIO
+    select VIRTIO_NET
+    select NET
     default KUNIT_ALL_TESTS
```

Изменения в файле Kconfig

При запуске можно наблюдать уже знакомую картину 3. Теперь все готово для имплементации тестов.

4.4 Технические детали разработки тестов

Целевым кодом тестирования является функция инициализации virtnet_probe драйвера virtio_net. Каждый из тест-кейсов вызывает функцию

virtnet_probe перед этим корректно сконструировав входные аргументы функции. Для конструирования корректных входных аргументов нужно учитывать особенности кода virtnet_probe и кода ядра.

Целевой код тестирования получает на вход так называемое виртуальное устройство, проверяет какие у него есть включенные свойства и инициализирует все нужные сущности связанные с этими свойствами. Свойства не являются частью виртуального устройства, они являются битами в целочисленном поле у структуры виртуального драйвера. Структура виртуального устройства хранит указатель на виртуальный драйвер. Тесты нацелены на увеличение объема покрытия кода, поэтому основная разница между тесткейсами это набор свойств, которые они включают. Список всех поддерживаемых свойств можно получить макросом **VIRTNET_FEATURES** из файла с кодом драйвера.

Кроме свойств нужно инициализировать набор разных функций указатели на которые хранятся внутри структуры virtio_config_ops. Эти функции нужны для инициализации драйвера и дальнейшей работы с подключенным драйвером. Указатели на эти функции хранятся внутри виртуального устройства, это облегчает жизнь для написания новых драйверов, где можно легко переопределить функции под свои нужды. В модульных тестах подавляющее большинство функций имплементированы как заглушки, которые приводят к желаемому поведению. Но есть исключения одним из которых является функция find_vqs.

find_vqs это основная функция, внутри тела которого можно получить доступ к сущностям, которые были созданы внутри функции инициализации virtnet_probe. Следовательно, все основные выделения памяти и захват ресурсов должен происходить именно внутри тела данной функции. К таким выводам можно прийти изучив код драйверов, которые переопределяют данную функцию под свои нужды. Такими драйверами являются например

- drivers/virtio/virtio_vdpa.c,
- drivers/virtio/virtio_mmio.c,
- arc/um/drivers/virtio_uml.c.

Важным шагом является инициализация списка виртуальных очередей, указатель на которую принимает функция find_vqs. Для решения данной задачи нужно учитывать некоторые тонкости кода ядра. Основных особенностей два, первый заключается в том, что если структура хранит указатель на другую структуру, то в большинстве случаев где-то уже должна быть инициализированная соответсвующая структура. Вторая особенность связана с практикой наследования структур и динамического приведения наследников к родителям. Так как в конструкциях языка С наследования нет, в ядре оно реализовано на уровне кода. Если структура хранит другую структуру, то она является ее наследником и существует специальная функция contain_of которая вычисляет все нужные смещения внутри структур и возвращает указатель на родительскую или дочернюю структуру в зависимости от использования.

Внутри кода драйвера есть специальная структура virtnet_info, где хранятся указатели не все нужные структуры. Среди них есть виртуальные очереди для отправления и получения данных. Эти структуры на самом деле являются родительской структурой другой структуры, которая называется vring_virtqueue. Следовательно, нужно сначала создать виртуальное кольцо виртуальных очередей и уже внутри структуры кольца создать виртуальные очереди и их использовать в качестве очередей отправки и получения. Если инициализировать эти структуры как просто виртуальные очереди virtqueue, то при дальнейшей работе программы можно получить ошибку сегментирования памяти вызванным приведением типов и обращением к полям дочерней структуры.

Тут нужно добавить замечание, что некоторые структуры, константы и функции используемые внутри реализации функции find_vqs объявлены внутри модулей компиляции, а не заголовочных файлов. Следовательно, получить к ним доступ внутри файла с тестами подключив заголовочный файл не получиться. Для решения данной проблемы в работе используются два подхода. Первый заключается в том, чтобы скопировать весь нужный код внутрь созданного для тестов заголовочного файла. А второй, более радикальный заключается во включении файлов с кодом внутри созданного заголовочного файла.

```
1  // Первый способ
2  #include "../../drivers/base/basec."
3
4  // Второй способ
5  // from virtio_ring.c
6  struct vring_desc_state_split {
7      void *data;     /* Data for callback. */
8      struct vring_desc *indir_desc;  /* Indirect descriptor,
   if any. */
9  };
```

Второй способ назван более радикальным, так как система сборки никак не помогает и все содержимое файла с кодом копируется внутрь заголовочного файла во время компиляции. Так как включенные файлы сами включают в себя множество других заголовочных файлов, то время компиляции замедлится, что не хорошо при реализации нескольких легковесных модульных тестов. Но у данного способа есть явное преимущество по сравнению с первым, а именно экономия времени разработки. Если решать проблему первым способом, то разработчику нужно будет вручную поискать все нужные объявления и скопировать их из разных файлов в один. Данные объявления могут одержать другие, с которыми нужно будет проделать аналогичную работу. В данной исследовательской работе был сделан выбор в пользу первого

способа.

Остальная часть кода реализации тестов заключается в присваивание разных значений полям разных, уже заранее инициализированных структур. Эти значения отличаются в тест-кейсах, так как они зависят от набора включенных свойств виртуального устройства. В основном выставляются значения размеров разных массивов и списков виртуальных очередей, количество свободных виртуальных очередей, индексы последних свободных очередей, различного рода индикаторы и параметры.

Фреймворк KUnit при запуске тестов не запускает ядро, и не инициализирует все данные общего состояния, в некоторых случаях нужно создавать заглушки для функций, которые не связаны с тестируемым кодом и драйвером. Один из таких примеров является функция у структуры kernelfs_node. Данная структура связана с кодом файловой системы и используется при инициализации драйвера. При определенном наборе свойств инициализация драйвера меняет общее состояние ядра. Эти изменения в свою очередь приводят к вызову функций из других модулей ядра. Данная особенность является узким местом и приводит к разным новым проблемам и большим временным затратам связанным с изучением кода других модулей ядра. Это сильно замедляет разработку тестов и повышает потребность использования отдельных отладочных и верифицирующих программ.

4.5 Практика мокирования функций

Модульное тестирование подразумевает разработку небольших, легковесных тестов, которые проверяют работоспособность конкретного модуля или функции не запуская при этом всю систему. Очень часто возникает проблема с тем, что модули и функции используют и меняют общее состояние системы. При тестировании таких модулей приходиться разрабатывать код поддерживающий общее состояние системы, что в случае ядра очень трудозатратно. Для решения подобного рода задач на практике используют моки-

рование или фиктивные объекты.

Моки это предварительно запрограммированные сущности разработанные с учетом ожиданий спецификаций вызовов, которые они, как ожидается, получат. Они могут генерировать исключение, если получают вызов, которого не ожидают, и проверяются во время верификации, чтобы убедиться, что они получили все ожидаемые вызовы. Корректно разработанные моки эмулируют ожидаемый в тестировании результат работы всей системы при этом не запуская всю систему.

Фиктивные или поддельные объекты имеют рабочие реализации, но обычно используют какой-то короткий путь, который делает их непригодными для производства. Они обычно не эмулируют корректное состояние системы, но создают состояние достаточное для исправной работы модульных тестов.

У обоих подходов одна основная цель, изолировать тестируемый код. У изоляции множество преимуществ. При неудачном завершении теста ошибки можно будет искать не во всей системе, а в маленькой ее части. Изоляция ускоряет сам процесс работы тестов, также ускоряет и упрощает разработку тестов. В фреймворке KUnit существует несколько способов мокирования функций [8].

- Добавление компиляционных конфигураций в реализацию функций.
- Изменение объявления функций с пометкой слабых символов компилятора.
- Использование конструкций из фреймворка.

Первый способ самый простой, но с большим количеством недостатков. Данный метод подразумевает создание новых конфигураций и включение разных частей кода при компиляции с разными конфигурациями. Если

для каждого тест-кейса нужны разные реализации, то придется создавать соответсвующее количество конфигураций и запустить каждый тест отдельно. Это приводит к большим временным затратам при разработке и при запуске тестов.

Второй способ использует линковку, чтобы при запуске вызывалась не оригинальная функция, а реализация разработчика. Данный способ реализуется при помощи атрибута weak. Компилятор во время стадии линковки найдет два определения, но отбросит тот, который будет помечен слабым. Это считается сложным подходом и без крепких знаний работы компилятора может привести к ошибкам и проблемам. Большим недостатком является отсутствие запуска нескольких тестов одновременно, если они используют разные заменяющий оригинал реализации. Придется для каждого тест-кйса редактировать файл с тестами и заново скомпилировать, что крайне неэффективно.

Третий способ в отличии от первых двух использует конструкции разработанные в самом фреймворке, а не возможности компилятора. Поэтому тут учтены недостатки предыдущих подходов. Реализуется она следующим способом.

```
1    void func_to_mock(const char *str) {
2        KUNIT_STATIC_STUB_REDIRECT(func_to_mock, str);
3        /* real implementation */
4    }
```

Внутри тела функции добавляется специальный макрос, что позволит позднее во время работы тестов заменить вызов данной функции на вызов любой другой. Достигается это следующими конструкциями из фреймворка.

```
1    void mocked_version(const char *str) {
2        /* fake implementation */
3    }
4
5    void test_case(struct kunit *test) {
```

```
6         kunit_activate_static_stub(test, func_to_mock,
  mocked_version);
7         /* test implementation*/
8         kunit_deactivate_static_stub(test, func_to_mock);
9     }
```

Все подходы имеют общий недостаток, это редактирование исходного кода драйвера. Среди них только последний способ дает возможность использовать разные реализации фиктивных функций для разных тест-кейсов. Большим недостатком третьего способа является, то что он разработан и внедрен в ядро относительно недавно и не доступен на версиях ядра 5.10, 5.10-у и до 6.2.7 включительно.

4.6 Отладка кода

Отладка это процесс поиска и исправления ошибок в исходном коде любого программного обеспечения. Когда программное обеспечение работает не так, как ожидалось, разработчики изучают код, чтобы определить, причины разных ошибки. В процессе решения данной задачи используются средства отладки для запуска программного обеспечения в контролируемой среде, пошаговой проверки кода, анализа и устранения проблемы.

В качестве отладочной программы было принято решение использовать gdb [9]. gdb может выполнять четыре основных вида операций и другие действия в их поддержку, чтобы помочь выявлять ошибки на месте.

- Запустить программу, указав все, что может повлиять на ее поведение.
- Остановить программу при заданных условиях. Посмотреть или поменять значение любого регистра, переменной.
- Изучить, что произошло, когда программа остановилась.
- Изменить что-нибудь в отлаживаемой программе, чтобы поэкспериментировать с исправлением последствий одной ошибки.

Для отлаживания разработанных модульных тестов, нужно сперва скомпилировать тесты указав нужные параметры конфигурации. Нужно включить в файле .kunit/.kunitconfig следующие две настройки.

```
    CONFIG_KUNIT=y
    CONFIG_VIRTIO_NET_TEST=y
+   CONFIG_DEBUG_INFO_DWARF_TOOLCHAIN_DEFAULT=y
+   CONFIG_DEBUG_KERNEL=y
```

Изменения в файле .kunitconfig

После добавления недостающих конфигураций можно скомпилировать тесты следующей командой.

```
    ./tools/testing/kunit/kunit.py build
```

Далее нужно открыть отладчик gdb и запустить там бинарный файл vmlinux из директории .kunit. Таким образом можно отлаживать модульные тесты как обычную пользовательскую программу. Рекомендуется использовать отладчик gdb, так как это достаточно легковесный инструмент, не требующий дополнительных библиотек и интерфейсов. Для отладки достаточно обычного терминала. У ядра Linux существуют конфигурации для gdb, чтобы работать одновременно с такими опциями компилятора как -fsanitize=address и -static-libasan [9].

4.7 Работа с памятью

Фреймворк KUnit перед тем как запустить тесты инициализирует объекты и сущности, которые нужные непосредственно для ее работы. Поэтому перед тем как использовать какие-либо переменные или указатели, нужно аллоцировать память под соответствующие нужды. Выделять память обычными функциями аллокации, доступными в коде ядра, будет неудобно так как разработчику модульных тестов придется следить за освобождением памяти и это может с легкостью привести к утечке памяти. В фреймворке реализована своя функция выделения памяти. Ее интерфейс дает возможность

только аллоцировать память в рамках тест-кейсов. За освобождение можно не задумываться, в конце исполнения набора тестов память освобождается автоматическим образом.

```
static void test_case_0(struct kunit *test) {
    struct vring_virtqueue *cvq = kunit_kzalloc(test, sizeof(
   struct vring_virtqueue), GFP_KERNEL);

    cvq->split.vring.desc = kunit_kzalloc(test, sizeof(struct
   vring_desc), GFP_KERNEL);
    cvq->split.desc_extra = kunit_kzalloc(test, sizeof(struct
   vring_desc_extra), GFP_KERNEL);
    cvq->split.vring.used = kunit_kzalloc(test, sizeof(
   vring_used_t) * 2, GFP_KERNEL);
}
```

Пример выделения памяти в тестах

Проверку и верификацию корректной работы с памятью можно сделать запустив тесты заранее скомпилировав с адресным санитайзером. Фреймворк дает возможность использовать аналогичные доступные инструменты из ядра указав нужные опции в конфигурационном файле. При выявлении некорректного обращения к памяти вывод в терминале будет дополнен вспомогательной информацией от санитайзера для быстрого нахождения и исправления ошибки в коде. Для работы санитайзера нужно указать следующие настройки.

```
    CONFIG_KUNIT=y
    CONFIG_VIRTIO_NET_TEST=y
+   CONFIG_KASAN=y
+   CONFIG_KASAN_VMALLOC=y
+   CONFIG_KASAN_INLINE=y
+   CONFIG_KASAN_GENERIC=y
+   CONFIG_STACKTRACE=y
```

Изменения в файле .kunitconfig

4.8 Покрытие кода

Список литературы

1. KUnit - Linux Kernel Unit Testing. — ©The kernel development community. https://www.kernel.org/doc/html/latest/dev-tools/kunit/index.html.
2. Learn about virtio-networking. — October 31, 2022 Ariel Adam. https://www.redhat.com/en/blog/learn-about-virtio-networking.
3. Virtio basic building blocks. — September 9, 2019 Ariel Adam, Amnon Ilan, Thomas Nadeau. https://www.redhat.com/en/blog/introduction-virtio-networking-and-vhost-net.
4. Running kernel with QEMU. — October 4, 2019 DaeSeok Youn. https://medium.com/@daeseok.youn/prepare-the-environment-for-developing-linux-kernel-with-qemu-c55e37ba8ade.
5. User Mode Linux. — ©The kernel development community. https://www.kernel.org/doc/html/v5.9/virt/uml/user_mode_linux.html.
6. Kernel compilers. — ©The kernel development community. https://mirrors.edge.kernel.org/pub/tools/crosstool/.
7. Kconfig Language. — ©The kernel development community. https://www.kernel.org/doc/html/next/kbuild/kconfig-language.html.
8. Fakes and Stubbing and Mocks. — 2019 Google LLC. https://kunit.dev/mocking.html.
9. Debugging kernel and modules via gdb. — ©The kernel development community. https://docs.kernel.org/dev-tools/gdb-kernel-debugging.html.