Тестирование драйверов файловых систем в ОС Linux[1]

А. В. Цыварев, В. А. Мартиросян
tsyvarev@ispras.ru, vmartirosyan@gmail.com

Аннотация. В статье исследуется проблема тестирования драйверов файловых систем ОС Linux. По результатам рассмотрения существующих систем тестирования, применимых к драйверам файловых систем, формулируются требования к системе тестирования, способной вывести решение этой задачи на качественно новый уровень. В первую очередь, для этого необходимо помимо проверки стандартной функциональности файловых систем, обеспечить тестирование поведения драйверов в редко встречающихся ситуациях, таких как сбои в нижележащем устройстве хранения, условия нехватки памяти, а также выявлять ошибки, связанные с утечкой ресурсов.

Ключевые слова. Linux; файловые системы; драйвер; модуль ядра ОС Linux; тестирование

1. Введение

Операционные системы, основанные на ядре Linux, широко используются в мире. Они обеспечивают работу более 90% мощнейших суперкомпьютеров из международного рейтинга TOP500 и являются лидером рынка операционных систем для мобильных устройств в составе ОС Android. Как и во многих других операционных системах, понятие файловой системы в Linux является одним из ключевых.

Linux поддерживает множество файловых систем, включая не только разработанные специально для него (ext2, ext3, ext4) и характерные для других Unix-подобных ОС (Xfs, Btrfs, Jfs), но и, например, характерные для ОС Microsoft Windows (FAT32, NTFS). Каждая из этих файловых систем имеет свои достоинства, делающие ее предпочтительной в определенных областях применения.

В Linux ответственность за работу файловых систем и операций с ними разделена между подсистемой виртуальной файловой системы (Virtual File System, VFS) ядра ОС, которое транслирует запросы пользователя в

[1] Работа поддержана ФЦП "Исследования и разработки по приоритетным направлениям развития научно-технологического комплекса России на 2007-2013 годы" (контракт N 07.519.11.4024)

последовательность операций над некоторым универсальным низкоуровневым представлением файловой системы, и драйверами файловых систем, которые реализуют эти операции в соответствии с собственной архитектурой и форматом хранения данных на диске. Если код ядра, отвечающий за работу файловых систем, достаточно отлажен и надежен, то ошибки в драйверах файловых систем все еще распространены. Об этом можно судить, в том числе, по журналу изменений [1] в ядре Linux, в котором достаточно часто встречаются записи об исправлениях ошибок в этих драйверах.

Наиболее распространенным способом проверки корректности функционирования драйверов является тестирование.

Здесь следует учесть, что в ОС Linux драйверы устройств, в том числе и драйверы файловых систем, реализованы в виде модулей ядра. Код этих модулей, как и данные, находится в одном адресном пространстве с ядром, и выполняется в привилегированном режиме. В тоже время, код ядра устроен таким образом, чтобы корректно работать при вызове из пространства пользователя через механизм так называемых системных вызовов — специальных функций, при вызове которых меняется режим выполнения кода с обычного (характерного для пространства пользователя) в привилегированный (характерный для пространства ядра).

По этой причине, тестирование драйвера обычно устроено не как непосредственный вызов функций драйвера в предварительно созданном окружении, ибо корректно создать это окружение «вручную» в пространстве ядра очень сложно. Обычный же тест работает в пространстве пользователя и использует системные вызовы, которые в свою очередь вызывают те или иные функции драйвера файловой системы. При этом корректное окружение при вызове функций драйвера создается самим ядром. Более того, сами системные вызовы чаще всего вызываются из теста не напрямую, а через более высокоуровневый интерфейс, например, через функции библиотеки libc, которые представляют собой удобные обертки для системных вызовов.

Системные вызовы представляют собой обобщенный интерфейс к функциональности операционной системы. Поэтому, чтобы системный вызов в итоге развернулся бы в вызов функции нужного драйвера, ему нужно передать определенные параметры. Для драйвера файловой системы такими параметрами может быть путь к файлу, принадлежащему файловой системе, обслуживаемой этим драйвером, или дескриптор этого файла. На рис.1 показана стандартная схема тестирования драйверов файловых систем.

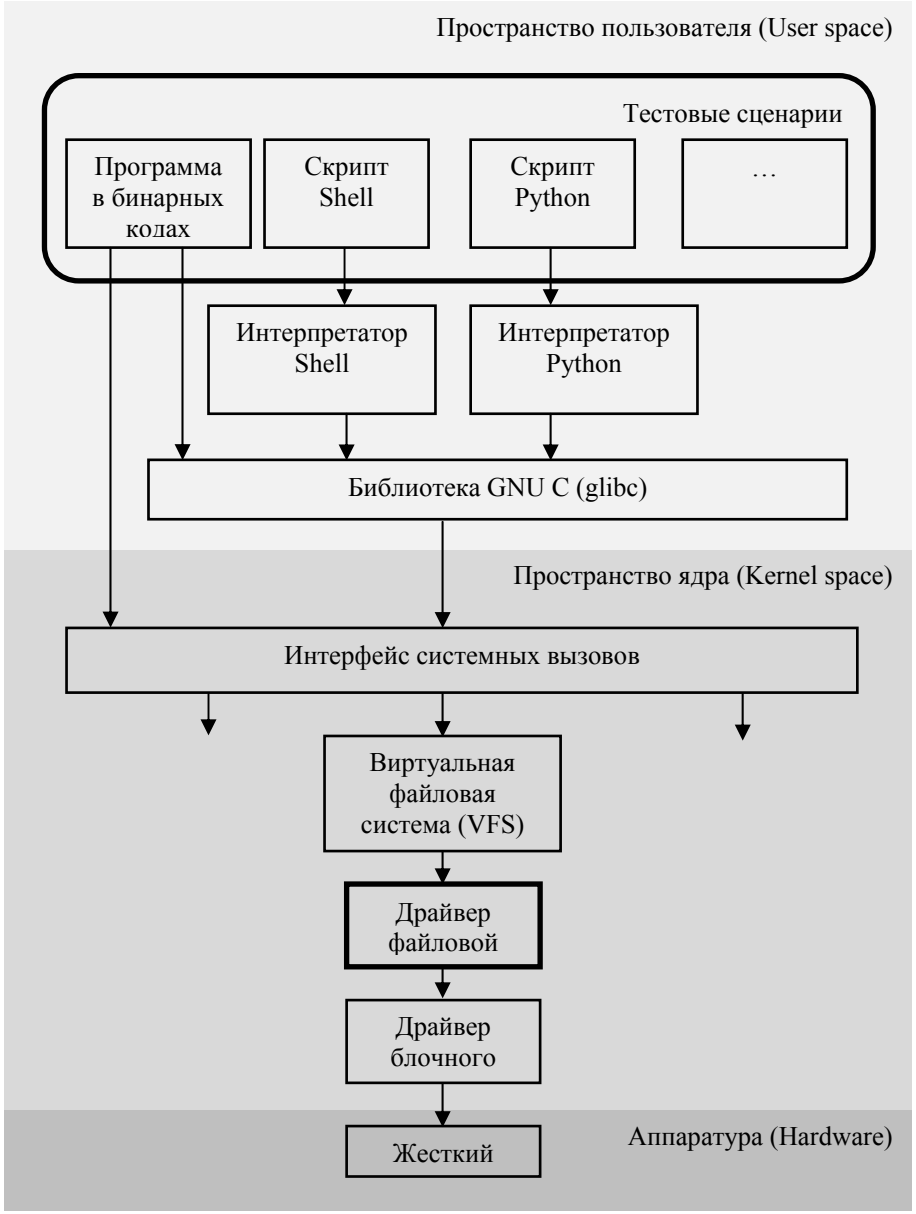

Рис. 1. Схема тестирования драйверов файловых систем.

Хотя за счет перебора параметров системных вызовов можно проверить большую часть функциональности драйвера, простого использования

системных вызовов с различными параметрами недостаточно для обеспечения качественного тестирования драйвера файловой системы. Во втором разделе статьи мы рассмотрим другие особенности устройства драйверов файловых систем ОС Linux. В третьем разделе мы покажем, какое влияние эти особенности оказывают на подходы к тестированию драйверов файловых систем. Далее будут рассмотрены существующие системы тестирования, используемые для поверки корректности функционирования драйверов, и сформулированы требования к системе тестирования, способной вывести решение этой задачи на качественно новый уровень.

2. Особенности устройства драйверов файловых систем

Рассмотрим особенности драйверов файловых систем, которые должны учитываться при их тестировании.

2.1. Реентерабельная программа с множеством входов

Как и многие другие типы драйверов, драйвер файловой системы представляет собой программу с множеством входов. Эти входы соответствуют функциям обратного вызова (callbacks), зарегистрированных определенным образом для обработки операций над файлами, директориями и другими объектами файловой системы (точнее, объектами ее низкоуровнего представления).

Отличительной особенностью драйверов файловых систем является очень большое количество таких функций: один драйвер может реализовывать до нескольких десятков функций, которые вызываются из ядра в и являются частью выполнения определенных системных вызовов. Более того, при выполнении некоторых системных вызовов вызывается не единственная функция драйвера, а целое множество, причем в определенной последовательности. Например, для системного вызова open() сначала ищется файл, соответствующий этому пути, если не найден — то файл создается, если найден — то для него создается и инициализируется дескриптор, и все эти операции выполняются за счет вызова функций драйвера.

Кроме того, драйверные функции обратного вызова могут вызываться из разных процессов, которые в свою очередь могут выполняться параллельно на различных ядрах процессора. Хотя подсистема виртуальной файловой системы VFS реализует определенную часть синхронизации между вызовами функций драйвера, основная нагрузка по обеспечению синхронизации ложится на драйвер файловой системы и занимает значительную часть его кода. Как показал анализ исправлений ошибок в стабильных ветках ядра за последний год [2] проблемы синхронизации в драйверах составляют самую большую долю ошибок (около 20%).

2.2. Работа в редких ситуациях

Основным сценарием использования драйвера является его работа при «нормальных» условиях. При таких условиях драйвер в состоянии выполнять свое основное предназначение; для драйверов файловых систем это может быть: создание файлов и директорий, чтение из файла и запись в него, получение различного рода информации о файловой системе и т. д. Помимо того, что это основной сценарий использования драйвера, это и наиболее частый сценарий.

Вместе с тем, иногда встречаются ситуации, когда выполнение драйвером основной задачи невозможно. Формально можно считать, что эти ситуации встречаются в ходе выполнения нетипичных или «ненормальных» сценариев использования драйвера. Такие сценарии встречаются относительно редко, поэтому и такие ситуации мы будем называть «редкими».

Одним из примеров таких редких ситуаций является вызов функций драйвера с некорректными пользовательскими данными. Хотя такой способ использования часто является следствием некорректной пользовательской программы, поведение драйвера в таком случае строго специфицировано, и драйвер обязан следовать этой спецификации.

Другой вариант редких ситуаций — нештатные условия, в которых выполняются функции драйвера. Под нештатными условиями чаще всего понимается состояние ОС, в котором часть ее возможностей не может использоваться (faulty environment). Для драйвера файловой системы нештатным состоянием ОС, препятствующим выполнению драйвером основных задач, может быть:

1. Нехватка свободной памяти в системе, или других ресурсов.
2. Невозможность писать/читать с устройства, на котором развернута файловая система (например, по причине сбойного жесткого диска)
3. Некорректное состояние файловой системы на устройстве, например, некорректная таблица файлов.

Требование корректной работы в нештатных ситуациях - это одно из принципиальных отличий компонентов ядра, к которым относятся и драйверы, от пользовательских приложений и библиотек.

Если пользовательская программа обнаруживает, что ее окружение не позволяет ей функционировать как положено, то обычная ее реакция — немедленное завершение работы. В некоторых случаях, возможны дополнительные шаги перед завершением работы: например, когда программа работает с важными данными, и их потеря или повреждение очень нежелательны, она может попытаться сохранить эти данные, или привести в согласованное состояние.

Ситуация меняется, когда речь идет о ядре и его модулях. Эти компоненты операционной системы обязаны корректно работать в любых ситуациях. Если

состояние системы не позволяет последней выполнить основную операцию, то функция должна корректно информировать об этом вызывающую сторону, при этом оставляя драйвер в согласованном и работоспособном состоянии. Для драйвера файловой системы реакцией на нештатные условия может быть:

1. Возврат файловой системы к состоянию, которое она имела до вызова функции драйвера, и возврат индикатора ошибки. Это самая «безобидная» реакция, и поэтому считается желательной реакцией драйвера на ситуацию, когда его функция не может выполнить свою основную задачу.
2. Возврат результата, наиболее согласующегося с состоянием файловой системы. Например, если какие-то данные дублируются, то в случае невозможности извлечь оригинал данных можно попробовать извлечь их дубликат и обработать его.
3. Перемонтирование файловой системы с опцией «только на чтение». Это обычная реакция драйвера в случаях, когда не удается записать данные, но вернуться в начальное состояние невозможно (например, удалось записать данные в файл, но не удалось отразить этот факт в журнале операций над файловой системой).

3. Особенности тестирования драйверов файловых систем

Принимая во внимание особенности устройства драйверов файловых систем, описанных в предыдущем разделе, рассмотрим, как это отразится на тестировании этих драйверов.

3.1. Тестирование параллельного выполнения кода драйвера

Так как код драйвера может выполняться параллельно, эту возможность надо проверять в тестах. Непосредственно проверить работу драйвера во всех ситуациях параллельного выполнения его кода практически невозможно из-за огромного количества таких состояний. Это обычная особенность тестирования параллельных программ. Одним из способов преодолеть это ограничение является анализ трассы выполнения кода драйвера на предмет так называемой конкуренции данных (data races).

3.2. Тестирование поведения драйверов файловых систем в редких ситуациях

Если редкая ситуация заключается в некорректных параметрах, переданных в функцию драйвера из пространства пользователя, то достаточно использовать соответствующий системный вызов, передав ему некорректный (с точки зрения обычной функциональности) параметр.

Если же редкая ситуация заключается в нештатном состоянии системы, то тестирование работоспособности драйвера в такой ситуации возможно несколькими способами. Наиболее прямолинейный подход для реализации тестирования этого требования — приведение системы к такому нештатному состоянию и последующий запуск тестовых сценариев.

Однако в случае нехватки свободной оперативной памяти такой подход не позволяет достичь качественного тестирования драйвера. Дело в том, что при системном вызове управление попадает в драйвер не сразу, а через промежуточные функции ядра. А эти функции, вообще говоря, также зависимы от наличия свободной памяти. И если в системе глобальная нехватка памяти, то очень вероятно, что управление даже не дойдет до тестируемого драйвера, а будет возвращена ошибка, например, о невозможности выделить память под дескриптор файла, что выполняется как раз промежуточной функцией ядра. В реальности, однако, возможна ситуация, когда нехватка памяти возникнет перед самым вызовом функции драйвера (если эту нехватку вызвал другой процесс, работающий параллельно).

Одним из способов протестировать поведение драйвера в ситуации нехватки памяти сохраняя при этом воспроизводимость сценариев тестирования, является эмуляция нехватки памяти в функциях, вызываемых непосредственно из драйвера. Так как все остальные функции ядра работают в «нормальных условиях», то управление из них всегда будет передаваться функциям драйвера (что важно, по предсказуемому сценарию). А сами функции драйвера уже будут «чувствовать» нехватку памяти, и должны показывать соответствующую реакцию. Для применения такого способа надо тем или иным способом работать в пространстве ядра (чтобы реализовать эмуляцию нехватки памяти).

Тестировать драйвер файловой системы в случае сбойного блочного устройства также лучше с использованием эмуляции.

3.3. Дополнительные проверки драйвера при тестировании

Не всегда по результату системного вызова можно определить, корректно ли себя ведет драйвер или нет. Примером некорректного поведения, ненаблюдаемого непосредственно из пространства пользователя, является утечка ресурсов ядра. Под утечкой ресурсов здесь подразумеваются запрошенные программой (пользовательским приложением, модулем ядра и пр.) ресурсы, которые с определенного момента не используются программой, но и не отдаются назад системе. Одним из часто используемых драйвером ресурсов является память, поэтому далее будем рассматривать именно ее, хотя большинство рассуждений применимы и к другим ресурсам.

Современные компьютеры обычно обладают большим объемом оперативной памяти, а поэтому ее небольшие потери чаще всего никак не влияют на работоспособность системы. Тем не менее, ситуация с утечками памяти не может считаться корректной для драйвера. Во-первых, даже если в результате

одной операции утечки памяти незначительны, то будучи выполненной много раз, операция может привести к значительной потери памяти, что может замедлить работу операционной системы, а то и вовсе привести к ее неработоспособности вследствие нехватки памяти, приводящей к перезагрузке. Во-вторых, сам факт «нецелевого» использования ресурсов не может считаться хорошим качеством драйвера.

Следовательно, драйвер не должен допускать утечек памяти, и это свойство стоит проверять при тестировании.

4. Обзор существующих систем тестирования, применимых к драйверам файловых систем

Задача тестирования Linux не нова, и существует немало систем, предназначенных для верификации тех или иных частей этой операционной системы. Рассмотрим некоторые наиболее популярные системы тестирования для ОС Linux, которые можно применить к файловым системам.

4.1. Системы тестирования общего назначения

Autotest [3] – система для автоматизированного тестирования. Эта система была разработана в первую очередь для тестирования ядра Linux, и включает тесты для этой цели.

Linux Test Project (далее LTP) [4] – проект, также направленный на тестирование Linux и его ядра.

Phoronix Test Suite [5] – платформа для оценки производительности различных компонентов Linux.

OLVER[6] - система для глубокого тестирования дистрибутивов на основе Linux на соответствие стандарту LSB Core.

Каждая из этих систем тестирования состоит из подсистемы управления и запуска тестов, и некоторого множества тестов и их наборов. Часть из этих тестов/наборов разработаны специально под соответствующую систему тестирования. Другие являются сторонними тестами/наборами, приспособленными к запуску под данной системой.

Исследуем тестовые сценарии из этих систем, которые применимы к файловым системам и их драйверам. В большинстве таких сценариев обычно выполняется некоторая характерная операция с файлами и директориями. Например, копирование файла, запись больших файлов и т. д. Другие сценарии заключаются в проверке работы того или иного системного вызова, который связан с файлами/директориями. Некоторые сценарии (например, compilebench в Autotest) проверяют работоспособность системы в случае параллельной работы с файлами. Но делают это на базовом уровне, просто запуская несколько потоков, выполняющих сходные действия.

В системе тестирования LTP можно включить режим тестирования, при котором симулируется общесистемная нехватка памяти и сбойное блочное устройство. Сценарий такой эмуляции чисто вероятностный — любой запрос памяти в ядре можно вернуть ошибку с константной вероятностью, тоже самое к запросам к блочному устройству. При этом сами тесты ничего не знают про такую эмуляцию, поэтому их вердикт «ошибка»(fail) теряет достоверность, а значит в таком режиме тестирования можно проверить только отказоустойчивость драйвера: упал/ не упал.

Минус вероятностного сценария симуляции ошибок заключается в отказе от точной воспроизводимости тестов (это отмечается, например, в статье [7]). Кроме того, в текущей реализации эмуляции системных ошибок в LTP эмуляция ошибок производится не только для тестируемого модуля, но и для всех других модулей и самого ядра. Это ведет к увеличению числа прогонов теста, необходимых для проверки обработки ошибок в тестируемом модуле: меньше вероятность, что ошибка будет симулирована именно в тестируемом модуле, и возможность ситуации, когда из-за симуляции ошибки в другом модуле, некоторый код из тестируемого модуля вообще не будет выполнен.

4.2. Сертификационные системы

Помимо систем тестирования, применимых к Linux в целом, также существуют системы тестирования предназначенные для запуска на определенных ОС, основанных на Linux. Такие системы в основном разрабатываются для сертификации программного обеспечения и физического оборудования на соответствие определенному дистрибутиву Linux. Рассмотрим наиболее известные сертификационные системы:

SUSE YES Certified Program [8] (в прошлом — Novel Yes Tools) – программа сертификации физического оборудования на совместимость с линейкой продуктов SUSE, в том числе SUSE Linux Enterprise, SUSE Open Enterprise Server.

Red Hat Hardware Program [9] – программа сертификации физического оборудования на совместимость с Red Hat Enterprise Linux.

Oracle Hardware Certification Program [10] – программа сертификации физического оборудования и систем на совместимость с ОС Oracle Linux.

Certification Program в Canonical [11]– программа сертификации физического оборудования на совместимость с ОС Ubuntu и поддержки качества.

В этих сертификационных программах используются как внешние тестовые системы и наборы (в том числе и упомянутые выше), так и специально разработанные для них.

Сертификационные системы для Red Hat и Canonical тестируют поведение оборудования, а следовательно и драйвера для этого оборудования, только в нормальных условиях. Две другие сертификационные системы проверяют

поведение драйвера также в случае нехватки памяти. Причем ситуация нехватки памяти создается в них разными способами.

В Oracle Hardware Certification Program ситуация нехватки памяти создается предварительным «отъемом» этой памяти у системы. Как уже писалось при рассмотрении тестирования при нехватке памяти, при таком подходе, теряется нацеленность тестирования на конкретный драйвер.

В SUSE YES Certified Program ситуация нехватки памяти симулируется только для тестируемого драйвера. Это достигается путем подмены вызовов функций, выделяющих память, из кода драйвера. Эти вызовы подменяются на вызовы специальных функций-оберток, которые с определенной вероятностью возвращают ошибку даже не пытаясь выделить эту память. Как и в случае с LTP, вероятностный сценарий симуляции ошибок ведет к меньшей воспроизводимости тестирования и к большему количеству прогонов теста.

Помимо симуляции нехватки памяти, в SUSE YES Certified Program отслеживаются операции запросов памяти и ее освобождения тестируемым драйвером. После завершения тестов и выгрузки драйвера из ядра проверяется, что все запрошенные драйвером куски памяти были им освобождены. Если это не так, то все неосвобожденные участки интерпретируются как утечки памяти. Из минусов этой реализации проверки утечек памяти стоит отметить небольшое количество перехватываемых функций (из-за этого запрос памяти через некоторые механизмы не отслеживается), и неотчуждаемость реализации от самой системы сертификации.

4.3. Системы тестирования нацеленные на определенные файловые системы

Характерным примером системы, нацеленной на тестирование драйверов файловых систем определенного типа, является Xfstests [12]. Эта система тестирования разрабатывается в рамках проекта по реализации на Linux самой файловой системы Xfs. Помимо файловой системы Xfs, эта система тестирования может применяться и для некоторых других журналируемых ФС, например, Btrfs и Ext4. Тесты в этой системе используют специальные последовательности операций с файлами, которые заставляют драйвер выполнять код, специально предназначенный для такого использования. На целевых файловых системах такая система обеспечивает гораздо более качественное тестирование, чем в системах тестирования общего назначения. Например, на дистрибутиве Debian 6 (ядро 2.6.32) покрытие по строкам кода драйвера Xfs составляет примерно 77%, тогда как в системах тестирования общего назначения это покрытие составляет порядка 50%.

5. Выбор требований и путей реализации системы тестирования драйверов файловых систем

На основе материала, рассмотренного выше, можно сформулировать требования к системе тестирования, которая обеспечивала бы более тщательное тестирование драйверов файловых систем, и предложить некоторые способы реализации этих требований.

5.1. Тестирование нормальной функциональности драйвера

Безусловно, функциональность драйвера в обычных условиях должна быть проверена. Тестирование этой функциональности может быть реализовано с использованием системных вызовов, параметры для которых подбираются такими, чтобы вынудить драйвер выполнить тот или иной код. Системные вызовы могут вызываться косвенно — через функции библиотеки libc, через команды shell, и т. д.

5.2. Тестирование работы драйвера в параллельных процессах

В сценариях тестирования должна быть также заложена работа драйвера в параллельных процессах. В качестве дополнительной проверки могут быть использованы подходы на основе проверки трассы выполнения кода модуля на предмет конкуренции данных (data races). В ИСП РАН в процессе разработки находится несколько инструментов, которые могут быть применены для выполнения такого рода проверок.

5.3. Тестирование системных вызовов с некорректными параметрами

В результате таких системных вызовов будут вызываться функции драйвера и будет проверяться их устойчивость к некорректным данным. Здесь стоит отметить, что некоторые некорректные данные проверяются и откидываются не доходя до вызова функций драйвера.

5.4. Тестирование работы драйвера при нехватки памяти

Один из эффективных способов такого тестирования — симуляция нехватки памяти в функциях, вызываемых из тестируемого драйвера. Такая симуляция может быть реализована с использованием инструмента fault injection [13], включенного в ядро ОС Linux, или KEDR Fault Simulation, разработанного в рамках проекта KEDR [14] в ИСП РАН.

5.5. Тестирование работы драйвера в случае сбойного блочного устройства

Драйвер файловой системы должен быть устойчив к сбойному блочному устройству, на котором развернута файловая система. Один из эффективных

способов такого тестирования — симуляция такого устройства в функциях, вызываемых из тестируемого драйвера. Для этого может быть использован как инструмент fault injection, так и KEDR Fault Simulation. В последнем случае такого рода симуляция должна быть реализована дополнительно, в стандартную поставку KEDR она не входит.

5.6. Тестирование работы драйвера в случае некорректного состояния файловой системы

Один из эффективных способов такого тестирования — иметь набор файлов с образами некорректных файловых систем, или иметь способ генерации такого набора образов. В качестве генератора образов некорректных файловых систем может быть связка утилиты dd, позволяющей перезаписывать любые участки блочного устройства, с утилитой, позволяющей осуществлять навигацию по элементам файловой системы. Утилиты для навигации есть для большинства широко используемых файловых систем: debugfs для Ext2/Ext3/Ext4, xfs_db для Xfs, jfs_debugfs для Jfs и т. д. Для некоторых файловых систем существуют готовые утилиты для направленной «порчи» файловых систем: fswreck [15] для OCFS, xfs_db[16] (команда blocktrash) для Xfs. Эти утилиты работают по сходным принципам — меняют некоторые байты/биты в заданном элементе файловой системы. Отличие состоит в том, fswreck модифицирует только элементы с заданными номерами, а xfs_db позволяет модифицировать сразу несколько элементов, выбрав их номера по псевдослучайному алгоритму с заданными параметрами.

5.7. Проверка утечек памяти в драйвере

Во время тестирования драйвера должна выполнятся проверка на утечки памяти. Для такой проверки подходит инструмент kmemleak(включен в ядро) [17]. Также можно воспользоваться инструментом KEDR Leak Check.

6. Заключение

Существующие тестовые наборы обеспечивают неплохой уровень тестирования, по крайней мере, наиболее распространенных драйверов файловых систем. Уровень покрытия в 77% строк кода драйвера XFS заметно выше среднего уровня покрытия по коду как в индустрии, так и среди компонентов ядра ОС Linux. Тем не менее, остаются сложности с обеспечением покрытия значительной части кода драйверов файловых систем ОС Linux, которые ввиду их особенностей требуют реализации целого комплекса специальных методов и подходов. В настоящей статье были рассмотрены эти особенности, сформулированы требования к формированию такого комплекса и предложены возможные направления решения поставленных задач.

Список литературы

[1]. Linux kernel bugzilla, https://bugzilla.kernel.org, 01.10.2012(дата обращения)
[2]. В.С. Мутилин, Е.М. Новиков, А.В. Хорошилов. Анализ типовых ошибок в драйверах операционной системы Linux. Труды Института системного программирования РАН, 2012 г., том 22, стр. 349-374. DOI: 10.15514/ISPRAS-2012-22-19.
[3]. Autotest Framework, http://autotest.kernel.org.
[4]. Linux Test Project, http://ltp.sourceforge.net.
[5]. Phoronix Test Suite, http://www.phoronix-test-suite.com.
[6]. Open Linux VERification Project, http://linuxtesting.org/olver.
[7]. Subrata.M, Balbir S., Masatake Y., Putting LTP to test – Validating both the Linux kernel and Test-cases. http://ltp.sourceforge.net/documentation/technical_papers/Putting_LTP_to_Test.pdf, 2009.
[8]. SUSE YES Certified Program, http://www.novell.com/developer/yes/, 2012.
[9]. Red Hat Hardware Program, http://www.redhat.com/rhel/compatibility/hardware, 2012.
[10]. Oracle Hardware Certification Program, http://www.oracle.com/webfolder/technetwork/hcl/hcts/index.html, 2012.
[11]. Canonical's certification service, http://www.canonical.com/engineering-services/certification/hardware-certification, 2012.
[12]. Xfstests sources, http://oss.sgi.com/cgi-bin/gitweb.cgi?p=xfs/cmds/xfstests.git.
[13]. Fault injection capabilities infrastructure, http://www.mjmwired.net/kernel/Documentation/fault-injection/
[14]. KEDR Project, http://linuxtesting.org/kedr.
[15]. OCFS2 tools sources, https://github.com/jjzhang/ocfs2-tools.
[16]. XFS user utilities sources, http://oss.sgi.com/cgi-bin/gitweb.cgi?p=xfs/cmds/xfsprogs.git.
[17]. Kernel memory Leak Detector, http://www.mjmwired.net/kernel/Documentation/kmemleak.txt.

Testing of Linux File System Drivers

A.V. Tsyvarev, tsyvarev@ispras.ru, ISP RAS, Moscow, Russia
V.A. Martirosyan, vmartirosyan@gmail.com, RAU, Erevan, Armenia

Annotation. The paper investigates issues of Linux file system driver testing. Linux file system drivers are implemented as kernel modules, which works in the same address space as kernel core. For that reason, the driver should be very reliable. It should react adequately to incorrect file system images, to faults in operating system, etc. Also drivers should free all resources it requests as far as kernel have to work for long time without restart. Another important feature of file system drivers is that they are programs with multiple entry points, which may be executed simultaneously in respect with each other and with other kernel code.

Most of existing test systems verify only driver's behavior in normal situations by calling system calls, which are eventually dispatched by the kernel into the driver's functions. Some of them also check driver in concurrent scenarios but only at very basic level. Some test systems also verify driver's behavior under faulty environment, when request for memory allocation or disk read/write may fail. But fault scenarios used in those systems are probabilistic, which leads to problems with tests' reproducibility and requires to repeat tests many times to achieve better coverage. There is one test system, which checks for memory leaks in the driver under test.

The paper concludes by statement of requirements for more throughout file system driver testing. According to the requirements a test system have to cover the following aspects:

1. Normal scenarios on system calls level.
2. Parallel scenarios with additional checks for data races.
3. Fault scenarios with insufficient memory and faulty block devices using such techniques as fault injection.
4. Handling of incorrect file system images.
5. Driver testing on system calls with invalid arguments.
6. Check leaks of memory and other resources requested by driver under test.

Keywords. Linux; file systems; driver; Linux kernel module; testing

References

[1]. Linux kernel bugzilla, https://bugzilla.kernel.org, 01.10.2012(дата обращения)
[2]. V.S.Mutilin, E.M. Novikov, A.V. Khoroshilov. Analiz tipovykh oshibok v drajverakh operatsionnoj sistemy Linux [Analysis of typical faults in Linux operating system drivers]. Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol. 22, pp. 349-374 (in Russian). DOI: 10.15514/ISPRAS-2012-22-19.
[3]. Autotest Framework, http://autotest.kernel.org.
[4]. Linux Test Project, http://ltp.sourceforge.net.
[5]. Phoronix Test Suite, http://www.phoronix-test-suite.com.
[6]. Open Linux VERification Project, http://linuxtesting.org/olver.
[7]. Subrata.M, Balbir S., Masatake Y., Putting LTP to test – Validating both the Linux kernel and Test-cases. http://ltp.sourceforge.net/documentation/technical_papers/Putting_LTP_to_Test.pdf, 2009.

[8]. SUSE YES Certified Program, http://www.novell.com/developer/yes/, 2012.
[9]. Red Hat Hardware Program, http://www.redhat.com/rhel/compatibility/hardware, 2012.
[10]. Oracle Hardware Certification Program, http://www.oracle.com/webfolder/technetwork/hcl/hcts/index.html, 2012.
[11]. Canonical's certification service, http://www.canonical.com/engineering-services/certification/hardware-certification, 2012.
[12]. Xfstests sources, http://oss.sgi.com/cgi-bin/gitweb.cgi?p=xfs/cmds/xfstests.git.
[13]. Fault injection capabilities infrastructure, http://www.mjmwired.net/kernel/Documentation/fault-injection/
[14]. KEDR Project, http://linuxtesting.org/kedr.
[15]. OCFS2 tools sources, https://github.com/jjzhang/ocfs2-tools.
[16]. XFS user utilities sources, http://oss.sgi.com/cgi-bin/gitweb.cgi?p=xfs/cmds/xfsprogs.git.
[17]. Kernel memory Leak Detector, http://www.mjmwired.net/kernel/Documentation/kmemleak.txt.