Тестирование операционных систем

Герлиц Е.А., Кулямин В.В., Максимов А.В., Петренко А.К., Хорошилов А.В., Цыварев А.В.
{gerlits, kuliamin, andrew, petrenko, khoroshilov, tsyvarev} @ispras.ru

Аннотация. Работа операционной системы лежит в основе функционирования любой компьютерной системы. Сбои и ошибки в операционной системе сказываются на работоспособности системы в целом, поэтому к корректности и надёжности операционных систем предъявляются повышенные требования. Верификация и тестирование операционных систем осложняется целым букетом разнообразных обстоятельств — это и зависимость от аппаратуры, и массированный внутренний параллелизм, и традиционное богатство конфигурационных настроек, и вопросы устойчивости к действиям злоумышленников и к сбоям аппаратуры, и продолжительность непрерывного функционирования. В статье рассматриваются все эти особенности, описываются подходы и инструменты тестирования, разработанные в Институте системного программирования РАН, и представляется опыт их применения для тестирования операционной системы Linux, а также ряда операционных систем реального времени.

Ключевые слова: операционная система; тестирование на основе моделей; тестирование производительности.

1. Введение

Операционные системы (ОС) решают две взаимодополняющие задачи:

- организуют процесс работы многих приложений на одной ЭВМ, управляя разделением ресурсов ЭВМ между приложениями, а также защищая приложения друг от друга;
- предоставляют набор функций с целью создания удобной среды для работы пользователя и прикладных программ.

Ключевым компонентом ОС является ядро (рис. 1), к которому, как правило, относят код, выполняющийся в привилегированном режиме работы процессора. Ядро управляет всеми аппаратными ресурсами, доступными ОС, предоставляя возможность настройки политик доступа к ресурсам и разделения ресурсов, а также не позволяя приложениям нарушать эти политики. Иногда в угоду интересам оптимизации в ядро включают функциональность, которая не требует привилегированного режима

процессора. Примером такого включения может служить фильтрация сетевых пакетов в ядре ОС Linux.

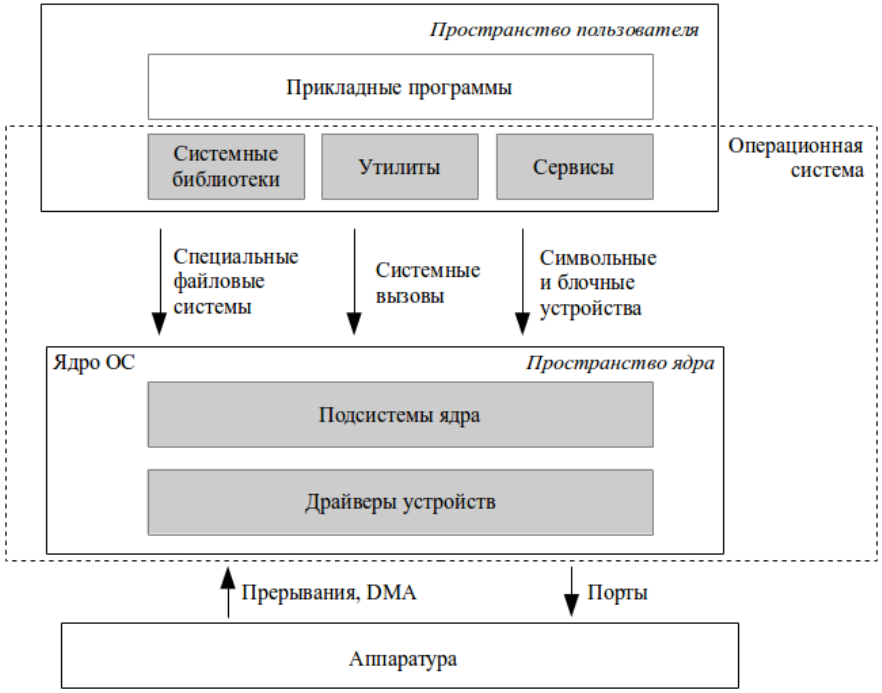

Рис. 1. Основные компоненты ОС.

Взаимодействие ядра ОС с пользовательскими приложениями обычно выполняется посредством системных вызовов, которые во многом схожи с обычным вызовом функции, но включают в себя переключение в привилегированный режим работы процессора и обратно. Часто также существуют и дополнительные механизмы взаимодействия с ядром, например, посредством чтения-записи файлов в специализированных файловых системах, таких как procfs, sysfs, debugfs в ОС Linux.

С целью создания удобной среды для работы прикладных программ ОС предоставляет системные библиотеки и утилиты, которые реализуют множество типовых функций и при необходимости обращаются к ядру ОС.

Ещё одним важным элементом ОС являются системные сервисы — активные компоненты, которые необходимы для реализации той или иной функциональности ОС. Например, для обслуживания определённых устройств, реализации сетевых протоколов, управления ресурсами и т. д.

Сервисы могут работать как внутри ядра ОС, так и в пользовательском пространстве.

Рассматривая ОС с точки зрения тестирования, необходимо учитывать следующие особенности.

- ОС как программное обеспечение, работающее непосредственно с аппаратурой, обладает:
  - значительным внутренним параллелизмом;
  - зависимостью от аппаратуры и её конфигураций;
  - внутренней активностью.
- ОС как основа компьютерной системы в целом и гарант безопасности приложений должна:
  - быть устойчивой к различным нестандартным ситуациям, таким как нехватка оперативной памяти, памяти жёсткого диска и т. д.;
  - быть устойчивой к атакам и вредоносным действиям со стороны недоверенных приложений, сетевых контрагентов, подключаемых внешних устройств и т. д.
  - не допускать утечек ресурсов с целью обеспечения продолжительного функционирования системы без перезагрузки;
  - минимизировать накладные расходы на реализацию своих функций.
- ОС как среда для работы прикладных программ должна обладать такими свойствами как:
  - соответствие стандартам на интерфейсы ОС;
  - соответствие документации на интерфейсы ОС;
  - совместимость с приложениями как на уровне бинарных интерфейсов, так и на уровне исходного кода.
- ОС как платформа, на которой работает тестовая система, является источником дополнительных требований к принципам построения тестовой системы:
  - при обнаружении ошибок, ведущих к аварийному прекращению работы ОС, информация, полученная тестом, должна сохраняться;
  - тестовая система должна вносить минимальные искажения в поведение тестируемых компонентов ОС, в том числе в их временны*е* характеристики.

Последующие разделы статьи организованы следующим образом. В разделе 2 рассматриваются вопросы функционального тестирования ОС, как тщательного, так и поверхностного. Раздел 3 посвящён тестированию обратной совместимости ОС, а раздел 4 — обнаружению специфических ошибок, таких как утечки ресурсов, гонки по данным и некорректная обработка некорректных входных данных. В разделе 5 рассматривается задача тестирования производительности ОС. В разделе 6 описаны сложности конфигурационного тестирования ОС и необходимость комплексной системы

управления тестированием ОС. В заключении подводятся итоги и формулируются направления дальнейшего развития.

2. Функциональное тестирование

Одной из ожидаемых характеристик ОС является корректная реализация своей функциональности. Это означает, что реальное поведение компонентов, доступных через публичные интерфейсы ОС, должно соответствовать декларируемым. Среди публичных интерфейсов ОС можно выделить интерфейсы системных библиотек и утилит, а также интерфейсы ядра ОС. Единичным элементом интерфейса библиотеки является функция, например, open(), sin() или fprintf(). Приведённые примеры демонстрируют различные виды реализации библиотечных функций:

- open() является примером функции-обёртки, которая сама по себе мало, что делает, за исключением трансляции запроса к компоненту более низкого уровня, в данном случае, open() обращается к соответствующему системному вызову ядра ОС;
- sin(), напротив, полностью реализуется в математической библиотеке и не требует обращения к другим компонентам;
- fprintf() представляет собой промежуточный вариант, в котором существенная часть функциональности реализуется в библиотеке (форматирование строки), но присутствует также и обращение к внешним компонентам (непосредственный вывод строки).

Таблица 1 демонстрирует количественные показатели состава интерфейсов в ОС. На одной стороне спектра фигурирует специализированная операционная система реального времени (ОСРВ), которая предназначена для работы на встраиваемых устройствах. Она включает поддержку порядка 700 функций и 80 команд, хотя при подготовке образа ОСРВ для конкретного применения в него включают только то подмножество функциональности, которое действительно необходимо для работы. В качестве другой стороны спектра представлена статистика ОС общего назначения Debian 7.0 для архитектуры x86, которая включает в себя более 10 тысяч утилит и более полутора тысяч библиотек, предоставляющих более 700 тысяч функций.

	Дата выпуска	Системные вызовы	Библиотеки	Функции	Утилиты
Debian 7.0	Май 2013	~350	~1650	~ 720 тыс.	10 тыс.
ОСРВ	Ноябрь 2013	~200	-	~700	~80

Табл. 1. Количественные характеристики размера интерфейса ОС.

Представленная статистика наглядно показывает, что задачу тщательного функционального тестирования можно ставить только для специализированных ОС и для ключевых интерфейсов ОС общего назначения. Для широкомасштабного функционального тестирования ОС общего назначения приходится использовать методы, в большей степени ориентированные не на достижение высокого качества тестирования, а на обеспечение приемлемых временны*х* и экономических показателей проектов.

2.1. Тщательное функциональное тестирование

Для тщательного функционального тестирования необходимым условием является наличие детальных функциональных требований к целевым интерфейсам, а также чёткое представление об архитектуре компонентов, реализующих целевую функциональность. При этом важно отметить, что целью тестирования является выявление дефектов не только в реализации ОС, но и в документах, описывающих функциональные требования и архитектуру.

2.1.1 Оценка качества тестирования

Поскольку исчерпывающее тестирование любой сколько-нибудь сложной системы не представляется возможным, то другим непременным атрибутом тщательного тестирования является формальная и всесторонняя оценка качества проводимого тестирования, которая позволяет оценить, насколько полно тестирование покрывает различные аспекты целевой функциональности. Для оценки качества тестирования традиционно используется два взаимодополняющих подхода:

- оценка покрытия функциональных требований;
- оценка покрытия структуры исходного кода реализации.

Для оценки покрытия требований необходимо, чтобы эти требования были структурированы, и минимальные элементы структуры представляли собой элементарные требования, то есть не подлежали дальнейшей декомпозиции. Кроме того, требования должны быть проверяемыми. Часто оказывается, что имеющиеся функциональные требования описаны в виде текста на естественном языке, допускают различные интерпретации и не обладают всеми нужными свойствами. В этом случае в процесс разработки тщательных функциональных тестов приходится включать дополнительный шаг — выделение и каталогизацию элементарных требований [1, 2]. Для более детального слежения за покрытием требований также бывает необходимо оценивать, в каких ситуациях каждое требование проверялось. Для каталогизации требований и отслеживания их покрытия в ИСП РАН был разработан специализированный инструмент управления требованиями Requality [3, 4], который позволяет построить каталог требований таким образом, чтобы каждое требование было связано с соответствующим фрагментом текста в исходном документе с функциональными требованиями.

Эта возможность оказывается наиболее востребована, когда исходные документы развиваются независимо от процесса тестирования и может потребоваться синхронизация каталога требований с новыми версиями исходных документов.

Измерение покрытия структуры исходного текста поддержано целым рядом инструментов. Основной вопрос, который возникает при применении этой метрики — это как очертить границы компонентов, за покрытием которых необходимо следить, и как относиться к непокрытым фрагментам кода. Для ответа на первый вопрос как раз и требуются знания об архитектуре компонентов, реализующих целевую функциональность. Второй вопрос подразумевает дополнительную активность, заключающуюся в анализе результатов измерения покрытия и принятии решений либо о доработке тестов, либо о внесении исправлений в исходный код, либо о признании отсутствия необходимости покрывать отдельные фрагменты.

Совместная оценка покрытия требований и покрытия исходного кода работают весьма эффективно, позволяя идентифицировать проблемы, которые могут остаться незамеченными при применении только одной из метрик, например, ввиду отсутствия в коде реализации одного из требований или отсутствия какого-то аспекта в требованиях.

Дополнительной метрикой, которой имеет смысл уделять внимание, является оценка покрытия возможных классов взаимодействия тестируемых компонентов с другими компонентами ОС [5].

2.1.2 Принципы построения тестов

Традиционный подход к построению тестов предполагает, что каждый тест представляет собой независимую единицу, состоящую из цепочки действий. Цепочка начинается с пролога, в котором происходит необходимая инициализация и подготовка системы к тестированию. Затем следует основная часть, содержащая активные воздействия на целевую систему, действия по наблюдению за поведением целевой системы и вынесение вердикта о корректности этого поведения. В заключении цепочки находится эпилог, в котором происходит освобождение ресурсов и приведение системы в исходное состояние. При построении тестов на основе функциональных требований, каждая проверка корректности обычно помечается идентификатором соответствующего элементарного требования.

Поскольку часто оказывается, что целая группа тестов выполняет приблизительно одни и те же действия, отличаясь лишь в небольшом числе деталей, то естественным развитием традиционного подхода является та или иная форма шаблонов тестов, позволяющая минимизировать дублирование кода. Примером такого развития является разработанная в ИСП РАН технология автоматизированной разработки тестов T2C (Template-to-Code) [6].

В основе T2C лежит автоматическая генерация исходных кодов тестов и других файлов, необходимых для компиляции и запуска тестов, из файлов,

содержащих шаблоны тестов. Эти файлы (далее - t2c-файлы) с шаблонами тестов создаются разработчиком тестов. Каждому тесту в t2c-файле соответствует своя секция, в которой задаётся тестовый сценарий на языке Си.

Проверки требований в тестовом сценарии выполняются с помощью специального макроса (REQ), которому в качестве параметров передаются:

- идентификатор проверяемого требования;
- булевское выражение на языке Си, невыполнение которого означает нарушение проверяемого требования;
- комментарий, который выводится в случае невыполнения проверки.

Тестовый сценарий может быть сам превращён в шаблон за счёт вынесения в параметры произвольных частей кода сценария, таких как входные данные, ожидаемые значения, идентификаторы проверяемых требований и т. д. После этого для тестового сценария можно указать один или несколько наборов значений параметров. В результате из одной секции тестового сценария в t2c-файле во время генерации кода может быть создано один или несколько тестов, каждый из которых получается подстановкой своего набора значений параметров.

Генерация кода тестов из t2c-файлов выполняется на основе специальных шаблонов генерации. За счёт выбора таких шаблонов можно получить то или иное представление тестового набора. По умолчанию генерируется набор тестов, пригодный для запуска под системой управления тестами TET [7]. Но есть также возможность генерации тестов, интегрируемых в другие системы выполнения тестов, а также так называемого автономного набора тестов, в котором каждому тесту соответствует отдельный исполняемый файл. В таком виде, например, можно сгенерировать отчуждаемый код для детального анализа конкретной ошибки.

Система T2C использовалась при разработке тестов для библиотек Linux, входящих в стандарт Linux Standard Base [8, 9]. С её помощью были подготовлены тесты для 10 библиотек. Более подробная статистика относительно тестирования части этих библиотек представлена в табл. 2. Описания найденных ошибок опубликованы на сайте [10].

Библио-тека	Версия	Протести-ровано интерфейсов	Проверено требований	Покрытие по коду	Найдено ошибок
libatk-1.0	1.19.6	222 из 222 (100%)	497 из 515 (96%)	-	11
libglib-2.0	2.14.0	832 из 847 (98%)	2290 из 2461 (93%)	12203 из 16263 (75.0%)	13
libgthread-2.0	2.14.0	2 из 2 (100%)	2 из 2 (100%)	149 из 211 (70.6%)	0
libgobject-2.0	2.16.0	313 из 314 (99%)	1014 из 1205 (84%)	5605 из 7000 (80.1%)	2
libgmodule-2.0	2.14.0	8 из 8 (100%)	17 из 21 (80%)	211 из 270 (78.1%)	2
libfonconfig	2.4.2	160 из 160 (100%)	213 из 272 (78%)	-	11
Всего		**1537 из 1553 (99%)**	**4033 из 4476 (90%)**	**18168 из 23744 (76.5%)**	**39**

Табл. 2. Результаты тестирования LSB библиотек тестами T2C.

Таким образом, система T2C продемонстрировала свои положительные стороны при достаточно масштабном применении, в условиях, когда не ставилась цель протестировать целевые библиотеки сверхтщательно и 80% покрытие считалось вполне достаточным.

В случае же, когда требуется более тщательное тестирование отдельных компонентов, наш опыт показывает [11], что наиболее эффективным подходом является разработка тестов на основе моделей. В рамках настоящей статьи мы продемонстрируем принцип построения тестов на основе моделей на примере технологии UniTESK [12], разработанной в ИСП РАН. В соответствии с технологией UniTESK функциональные требования представляются в формальном, машино-читаемом виде при помощи предусловий и постусловий модельных операций, а также инвариантов модельного состояния. Модельное состояние описывает представление тестовой системы о внутреннем состоянии целевой системы, а инварианты формулируют ограничения на его внутреннюю согласованность. Модельные операции, как правило, соответствуют интерфейсным функциям целевой системы. Предусловие описывает ограничения на состояние и аргументы функции, с которыми допустимо обращаться к ней, а постусловие формализует требования к результату работы функции, заключающемуся в

изменении состояния системы и формировании возвращаемого значения. При наличии каталога требований все проверки в инвариантах, предусловиях и постусловиях помечаются идентификатором соответствующего им элементарного требования из каталога.

Если в традиционном подходе к построению тестов проверка результата осуществляется вручную при помощи описания ожидаемого результата в явном виде, то при построении тестов на основе моделей наличие формальной модели требований позволяет выполнять такие проверки автоматически для произвольной тестовой ситуации. Как следствие, открывается возможность для автоматической генерации последовательностей тестовых воздействий на целевую систему, которые позволяют проверить целевую функциональность во множестве разнообразных ситуаций. Подготовить вручную сопоставимый массив тестов зачастую весьма непросто, а ещё сложнее такой массив сопровождать. Кроме того, наш опыт показывает, что автомат позволяет покрыть такие «угловые» случаи, которые вряд ли попадут в число тестов даже у очень изобретательного тестировщика. Достаточно часто в таких случаях находятся ошибки. В качестве примера, можно привести ситуацию с одной из реализаций очереди сообщений, в которой происходило зависание двух потоков, когда одновременно в списке ожидания на посылку сообщения в очередь и в списке ожидания на получение из очереди оказывалось более одного потока. А такое может случиться только в результате нетривиальной последовательности обращений к очереди из множества потоков с различными приоритетами.

Последовательность в рассматриваемом примере была сгенерирована при помощи метода построения нацеленных тестовых сценариев технологии UniTESK. Этот метод предлагает формировать тестовые сценарии, нацеленные на тщательную проверку определённой подобласти функциональности тестируемой системы на основе комбинации из трех элементов:

- модели требований;
- функции редукции модельного состояния;
- множества элементарных тестовых воздействий.

Модель требований описывает требования к целевой системе и не зависит от конкретного сценария. Функция редукции модельного состояния в совокупности с множеством элементарных тестовых воздействий формируют целеполагание данного тестового сценария, указывая, какие части модельного состояния являются важными, и подсказывая, из каких действий следует формировать последовательность тестовых воздействий. Далее метод предполагает, что инструменты, его реализующие, в ходе проведения тестирования автоматически сгенерируют последовательность тестовых воздействий таким образом, чтобы в каждом достигнутом редуцированном состоянии были выполнены все элементарные тестовые воздействия.

Ещё одним важным компонентом в технологии UniTESK являются медиаторы, которые устанавливают соответствие между моделью требований и тестируемой реализацией. Модельные операции отображаются на интерфейсные операции целевой системы, а модельное состояние обновляется медиатором на основе его представления о состоянии реализации.

С точки зрения тестирования ОС применение методов построения тестов на основе моделей, предполагающих достаточно много активностей в ходе самого тестирования, вызывает вопрос о возможных возмущениях, вносимых в работу ОС этими активностями — генерацией тестовых последовательностей и проверкой соответствия поведения тестируемых компонентов модели требований. Поскольку ОС управляет работой всех активностей на доверенном ей оборудовании, то для многих компонентов ОС такое возмущение может оказаться достаточно существенным.

С целью минимизации возможных возмущений мы предложили распределенную схему организации тестирования [13], согласно которой только небольшая часть тестовой системы работает на тестируемой ОС, а основная часть работает на отдельной инструментальной системе. Это достигается путём разделения медиатора на две части, которые взаимодействуют между собой через некоторый коммуникационный канал, например, через последовательный порт или сетевой интерфейс. Часть, работающая на тестируемой ОС, называется агентом. Её задача заключается в оказании тестовых воздействий в соответствии с указаниями, приходящими от тестовой системы, наблюдением за поведением целевой системы и передача собранной информации тестовой системе. Все остальные компоненты тестовой системы выполняются вне целевой ОС и никак не влияют на её работу. Таким образом, остаётся только обеспечить отсутствие влияния коммуникационного канала на целевые компоненты ОС.

Следует отметить, что многие интерфейсы ОС основаны на асинхронных принципах взаимодействия, что не вписывается в классический подход к описанию требований в виде предусловий и постусловий. Технология UniTESK была расширена для поддержки описания модели требований и построения нацеленных тестовых сценариев для систем с асинхронным интерфейсом [13].

Технология построения тестов на основе моделей для компонентов ОС применялась при тестировании ключевых компонентов, описываемых стандартами POSIX [14] и LSB [8, 9], и при тестировании ОСРВ на соответствие стандарту ARINC-653 [15]. Так, в проекте OLVER [16] целью тестирования были 1532 функции, описываемые стандартом LSB Core 3.1, в основном из библиотек glibc. В ходе анализа требований стандарта к этим функциям был составлен каталог требований, содержащий более 22 тыс. элементарных требований. Для поверки этих требований было разработано более 450 нацеленных тестовых сценариев, хотя ввиду ограничений по ресурсам далеко не для всех областей тестируемой функциональности

ставилась задача обеспечить полное покрытие требований. Поскольку тестируемый код поддерживал достаточно много расширений, не описанных в стандарте, на соответствие которому проводилось тестирование, то метрика покрытия по исходному коду не могла выступать в виде объективного критерия оценки качества тестирования на соответствие. В проекте тестирования ОСРВ [17] проверялись 54 функции, описываемые стандартом ARINC-653 часть 1, к которым предъявлялось 315 элементарных требований, и все они были покрыты тестами. Ввиду отсутствия доступа к исходному коду ОСРВ измерения покрытия по коду не проводились.

2.1.3 Тестирование обработки внутренних ошибок

Существенная часть кода многих компонентов ОС отвечает за обработку ошибок, которые могут возникнуть при выполнении различных операций, например, по причине нехватки памяти или сбоя при обращении к аппаратуре. Для того, чтобы проверить код обработки ошибок, требуется создать ситуацию, когда эти ошибки в действительности случатся. Причём создавать такие ситуации нужно целенаправленно, так как для сложных компонентов удар «по площадям» не работает. Например, попытки организовать «истощение» памяти в ядре ОС натыкается на сильное «сцепление» компонентов ядра, из-за которого чрезвычайно сложно создать ситуацию, когда нехватка памяти будет обнаружена именно тем компонентом, который тестируется, а не компонентом, который находится раньше в цепочке взаимодействия компонентов.

При тестировании кода, работающего в пользовательском пространстве, существует достаточное количество инструментов, которые позволяют решить данную проблему. Например, типичным подходом является использование специализированных библиотек выделения динамической памяти, которые реализуют стандартный интерфейс malloc/free и при запуске целевых компонентов подменяют стандартные библиотеки посредством возможностей, предоставляемых загрузчиком приложений ОС.

При тестировании компонентов ОС, работающих в пространстве ядра, задача не имеет такого простого решения. В первую очередь следует отметить, что хотя код ядра исполняется в адресном пространстве, отделенным от пространства пользователя, сценарии тестирования ядра обычно задаются с помощью программ, исполняющихся в пользовательском пространстве. Дело в том, что взаимодействие различных подсистем внутри ядра достаточно сложно, и написать тестовый сценарий, пригодный для исполнения полностью в пространстве ядра, весьма непросто. С другой стороны, большинство функциональности ядра (в том числе и драйверов), написана из расчёта их вызова при обработке того или иного системного вызова из пространства пользователя или других способов взаимодействия пользовательских программ с ядром ОС. Поэтому использование пользовательских программ в качестве сценариев тестирования ядра ОС даёт хорошие результаты.

Таким образом, для тестирования устойчивости компонентов ядра необходима возможность управлять сбоями в ядре из пользовательского пространства. Рассмотрим, какие инструменты для этого можно использовать на примере ОС Linux.

В коде ядра Linux присутствует инструмент Fault Injection [18], позволяющий симулировать сбой той или иной функции выделения памяти. За счёт различных опций можно добиться симуляции сбоя в том или ином вызове функции. Существенным для автоматического тестирования недостатком этого инструмента является ограниченность сценариев симуляции сбоев: инструмент поддерживает только вероятностный сценарий (при каждом вызове функции есть вероятность N, что вызов вернёт ошибку) и интервальный (задаётся минимальное время между последовательными симуляциями сбоев). С помощью таких сценариев очень сложно ограничить симуляцию сбоев только заранее заданным вызовом функции, из-за чего теряется воспроизводимость поведения системы при тестировании. Дополнительные сценарии можно реализовать только модифицировав код ядра.

Для преодоления обозначенных проблем в ИСП РАН была разработана платформа KEDR [19, 20], позволяющая наблюдать за модулем ядра ОС Linux и осуществлять перехват вызовов этим модулем внешних функций, таких как функции основной части ядра или функции, реализованные в другом модуле. Платформа реализована в виде модуля ядра, который за счёт инструментирования кода целевого драйвера позволяет выполнить код до, после или вместо вызова определённой внешней функции. Код для выполнения задаётся в дополнительных модулях, которые могут быть загружены в ядро при необходимости.

На платформе KEDR был реализован специализированный инструмент KEDR Fault Simulation, позволяющий симулировать сбои в функциях, вызываемых драйвером. В состав инструмента уже включены широкие возможности по организации сценариев симуляции, такие как возможность задавать номер вызова функции, который вернёт ошибку, указывать условия на значения параметров функции ("size >= 256 && flags != GFP_ATOMIC") и на адрес, откуда выполняется вызов. Это уже позволяет разрабатывать тесты с воспроизводимым поведением системы. Помимо этого, KEDR Fault Simulation позволяет реализовывать дополнительные сценарии без модификации кода ядра и самого инструмента, а также расширять список функций, для которых становится возможной симуляция сбоев.

В настоящее время KEDR Fault Simulation применяется в проекте по разработке тщательных тестов для драйверов файловых систем, работающих в пространстве ядра ОС Linux [21]. В ходе проекта инструмент продемонстрировал способность осуществлять нацеленную симуляцию ошибок при вызове функций ядра по указанию тестовых сценариев, работающих в пространстве пользователя. Помимо этого, в проекте был

предложен эффективный метод систематического внесения внутренних ошибок, который основан на следующей идее. Сначала в ходе выполнения традиционного функционального теста в специальном режиме выявляются все вызовы функций, которые могут возвращать ошибки. Затем каждый тест запускается N раз, где N – число таких вызовов функций в тесте. Каждый i-й запуск выполняется под управлением KEDR Fault Simulation со сценарием симуляции «i-й вызов функции возвращает ошибку». Этот метод продемонстрировал свою эффективность, позволив обнаружить несколько ошибок в таком зрелом драйвере файловой системе как ext4 и большое число ошибок в менее зрелых драйверах файловых систем.

2.1.4 Специализированные тестовые системы

Наш опыт разработки тщательных тестов для разнообразных компонентов ОС показывает, что часто для конкретного целевого компонента наиболее эффективным оказывается разработка специализированной тестовой системы, которая может базироваться на одном из рассмотренных принципов построения тестов, а может быть и полностью специфической.

Последний вариант, например, применялся при тестировании библиотек математических функций [22]. В этом случае каждая тестируемая функция получает одно или несколько чисел с плавающей точкой и должна вычислить число с плавающей точкой, наиболее точно представляющее результат соответствующей математической функции в текущем режиме округления и, возможно, выставить некоторые флаги, сигнализирующие об определённых событиях, таких как неточный результат, переполнение и т. д. При разработке тестов для таких функций наиболее трудоёмким является подбор множества тестовых значений, которые бы покрыли все значимые подпространства множества входных значений, интересные значения с точки зрения дилеммы составителя таблиц и т. д., а также вычисление при помощи нескольких библиотек функций, работающих с числами с повышенной точностью, корректных выходных значений для каждого входного значения. Инфраструктура же времени выполнения тестов достаточна проста: прочитать очередное входное и ожидаемое выходное значения, вызвать целевую функцию и проверить, насколько её результат отличается от ожидаемого. Нетривиальные составляющие появляются только при реализации автоматической аналитики обнаруживаемых ошибок, их кластеризации и т. д. Применение специализированного решения для тестирования математических функций продемонстрировало себя достаточно хорошо, поскольку позволило сосредоточиться на наиболее сложной подзадаче. А получившаяся тестовая система позволила обнаружить множество неожиданных результатов в распространённых библиотеках математических функций, в особенности, в нестандартных режимах округления [22].

Аналогичный математическим тестам подход к построению тестовых систем применялся и при тестировании ряда других компонентов ОС, таких как

обработка форматированного ввода-вывода. Также специализированные решения применялись при тестировании компонентов сетевого стека протоколов и драйверов файловых систем.

Если в случае тестирования протоколов за основу брался принцип построения тестов на основе моделей по технологии UniTESK [11], то при тестировании драйверов файловых систем ОС Linux в основе лежали идеи системы T2C, которые были адаптированы для тестирования компонентов ядра ОС.

Для тестирования драйверов файловых систем была создана специализированная система запуска тестов, которая поддерживает следующие возможности.

- Форматирование и монтирование тестируемой файловой системы перед началом теста. Большинство тестов реализованы как операции с файлами и директориями на уже примонтированной файловой системе.
- Опции форматирования и монтирования берутся из предварительно заполненного списка. Таким образом, каждый тест, который ориентирован на уже примонтированную файловую систему, выполняется несколько раз для разных вариантов форматирования и монтирования.
- Режим тестирования с систематической симуляцией сбоев, описанной в разделе 2.1.3.
- В случае, когда в результате одного из тестов система приходит в нестабильное состояние, в котором дальнейшее тестирование бессмысленно, есть возможность перезагрузить систему и продолжить тестирование. К таким необратимым состояниям относятся, например, срабатывания BUG_ON() в коде ядра или драйвера. Как показала практика, таких ошибок возникает немало, особенно при тестировании с симуляцией сбоев.
- Режим автоматического обнаружения утечек ресурсов с применением инструментов, описанных в разделе 4.1.

В табл. 3 представлено покрытие по исходному коду, которое достигается текущей версией тестов, и дополнительная добавка, получаемая за счёт активации метода систематической симуляции сбоев, описанного в разделе 2.1.3. В настоящий момент развитие тестового набора продолжается. Несмотря на то, что целенаправленный анализ обнаруживаемых тестами ошибок практически не проводился, в результате взаимодействия с разработчиками уже было исправлено около десятка ошибок в драйверах файловых систем, включая 4 ошибки в такой зрелой файловой системе как ext4.

Файловая система	Версия	Покрытие по строкам кода (без симуляции сбоев)	Покрытие по строкам кода (с симуляцией сбоев)
JFS	3.2	65%	71%
Ext4	3.2	60%	64%
F2FS	3.9	70%	75%

Табл. 3. Текущие показатели покрытия по строкам кода при тестировании драйверов файловых систем.

2.2. Поверхностное функциональное тестирование

Как демонстрирует статистика, приведённая в табл. 1, широкомасштабное функциональное тестирования библиотек ОС общего назначения тщательным образом не представляется возможным ввиду огромного количества имеющейся функциональности. Тщательное тестирование имеет смысл для ключевых компонентов или для компонентов, которые планируется применять для решения ответственных задач. Но это не означает, что тестировать остальные компоненты не нужно. В идеале за это должны отвечать разработчики соответствующих компонентов, но в сообществе разработчиков свободных программ это случается далеко не всегда. Большинство таких проектов просто не имеют необходимых ресурсов, чтобы уделять разработке тестов достаточное внимание, особенно на начальных стадиях. Да и по мере развития ситуация не становится проще, так как задача покрытия тестами всех функций библиотеки становится всё менее обозримой.

В данных условиях значительную помощь в исправлении ситуации могут оказать специализированные методы генерации поверхностных тестов или тестов работоспособности, под которыми обычно понимают тесты, проверяющие только то, что основные функции системы выполняются более-менее правильно, то есть, что система не разрушается и возвращает результаты, проходящие простейшие проверки на корректность (полная проверка при этом не выполняется).

Примером реализации таких методов являются инструменты Azov [23] и их дальнейшее развитие — API Sanity Autotest [24, 25]. Последний способен полностью автоматически генерировать тесты работоспособности только на основе заголовочных файлов библиотеки. При этом используется информация о сигнатуре публичных функций библиотеки, то есть о типах её входных и выходных параметров. Конечно, в большинстве случаев этой информации недостаточно для создания корректных тестов, например, из-за необходимости инициализации библиотеки или значений определённого типа неким нетривиальным образом.

Поэтому API Sanity Autotest поддерживает возможность задания дополнительной семантической информации об особенностях библиотеки, представленных в ней типах данных и специфики их использования в определённых функциях. Типичными примерами такой информации являются описания того:

- как получить корректное значение определённого типа данных;
- каким должно быть корректное значение определённого параметра функции;
- какие проверки можно сделать для возвращаемых значений определённого типа.

Может сложиться впечатление, что в этом подходе предлагается «варить суп из топора»: какая автоматизация, если требуется вручную описать код, необходимый и при обычной разработке тестов? Но тем не менее, польза от API Sanity Autotest есть как минимум по двум направлениям. Во-первых, описания пишутся в виде кода на C/C++, в котором могут присутствовать специальные конструкции, при помощи которых можно попросить инструмент сгенерировать код для получения значения определённого типа или подготовить параметры и вызвать определённую функцию. Во-вторых, семантическая информация одновременно привязывается ко многим местам, где она требуется. Например, информация о специфике инициализации библиотеки записывается один раз для всех функций. Информация о создании объектов определённого типа также записывается в одном месте и затем используется для всех функций, у которых есть параметр такого типа, и в других необходимых местах, таких как специальные конструкции, о которых шла речь выше.

Всё вместе это позволяет минимизировать дублирование кода, необходимого для подготовки описаний, что значительно сэкономит усилия при генерации тестов для больших библиотек, в которых типы данных и другие семантические элементы являются общими сразу для многих функций. Полученные в результате генерации тесты содержат минимальные проверки того, что функция работает на наиболее простом сценарии ее использования и возвращает более-менее правильный результат. Наличие таких тестов уже является большим шагом вперёд по сравнению с отсутствием тестов вовсе. И даже такие тесты позволяют обнаружить десятки ошибок.

Не менее ценным фактом является то, что сгенерированные тесты могут быть использованы в качестве хорошей стартовой точки для разработки полноценных функциональных тестов. Когда тесты работоспособности доведены до состояния корректной работы, то есть когда семантической информации достаточно для генерации корректных вызовов всех необходимых функций, полученный тестовый набор можно доработать вручную для обеспечения более качественного тестирования наиболее важных частей библиотеки. Для этих целей API Sanity Autotest поддерживает

генерацию тестов в формате T2C, который предоставляет удобные возможности для дальнейшего развития тестового набора.

Инструмент использовался для генерации тестов работоспособности для крупных библиотек, входящих в состав LSB: Qt3 (тесты для 9 792 функций), Qt4 (тесты для 10 803 функций), libxml2 (тесты для 1 284 функций). Кроме того, инструмент начал использоваться и разработчиками свободных библиотек.

3. Тестирование обратной совместимости

Современные операционные системы развиваются независимо от пользовательских приложений. Как следствие, возникает необходимость переноса приложений между различными версиями одной ОС. И так как приложений гораздо больше, чем ОС, то в большинстве случаев задача обеспечения переносимости перекладывается на плечи ОС в виде требования обеспечить обратную совместимость с приложениями. Это означает, что приложение, предназначенное для старой версии ОС, должно беспроблемно работать на новой версии ОС.

Обратная совместимость обычно рассматривается на одном из двух уровней: на уровне бинарных файлов или на уровне исходных кодов. В первом случае требуется, чтобы уже скомпилированное приложение можно было установить и использовать на новой версии ОС. Во втором случае речь идёт о возможности перекомпиляции исходного кода приложения под новую версию ОС без каких-либо изменений в коде.

Как правило, ОС обеспечивают обратную совместимость при условии, что приложения используют только специфицированные возможности ОС. Также существуют отраслевые и международные стандарты, регламентирующие интерфейс между приложениями и ОС, который позволяет обеспечить переносимость приложений между различными ОС. Примерами таких стандартов являются POSIX [14] и LSB [8, 9]. POSIX описывает интерфейс на уровне исходных кодов для семейства ОС UNIX. LSB нацелен на обеспечение переносимости на бинарном уровне между дистрибутивами ОС Linux.

Среди методов тестирования обратной совместимости можно выделить структурные и семантические.

3.1. Структурное тестирование обратной совместимости

При структурном тестировании обратной совместимости проверяется лишь наличие в новой версии ОС необходимой номенклатуры функций, которые входили в публичный интерфейс предыдущей версии, и, соответственно, не проверяется, что поведение этих функций осталось в рамках специфицированной ранее функциональности.

Примером инструментов структурного тестирования являются инструменты LSB libchk [26], который, имея список функций, входящих в состав стандарта

LSB, проверяет их наличие в соответствующих библиотеках тестируемой ОС. Поскольку в случае libchk проверка идёт на бинарном уровне ELF файлов, то для Си функций проверяется только имя функции и не отслеживаются изменения в её параметрах. Поскольку типы параметров для C++ функций кодируются в имена функций на бинарном уровне, то для них изменения в типах входных параметров могут быть обнаружены, но не могут быть обнаружены изменения в типе возвращаемого значения.

Схожим образом устроены сигнатурные тесты из Android Compatibility Test Suite [27]. Но они в исходных данных имеют список публичных интерфейсов с полной сигнатурой функции и, соответственно, проверяют неизменность сигнатуры в тестируемой версии ОС.

Следует отметить, что формирование и сопровождение списков публичных интерфейсов требует определённых усилий, и хотя оно хорошо вписывается в сертификационные программы, разработчикам библиотек такой подход не всегда удобен. Существует также ряд инструментов, позволяющих контролировать изменения в наборе функций, экспортируемых двумя версиями библиотеки, и, таким образом, избегать проблем, связанных с исключением функций. К таким инструментам относятся chkshlib [28], cmpdylib [29], cmpshlib [30], dpkg-gensymbols [31]. Все они работают по единому принципу, извлекая список функций из двух версий библиотек и сравнивая их. Различие состоит лишь в реализации этих инструментов. Другие виды несовместимостей эти инструменты находить не способны.

В качестве альтернативного варианта в ИСП РАН разработан инструмент ABI Compliance Checker [32], который в дополнение к сравнению списков экспортируемых функций позволяет находить несовместимые изменения в сигнатурах этих функций, в том числе опосредованные изменения, являющиеся следствием, например, изменений в определении соответствующих структур данных. Реализуется эта возможность за счёт дополнительного анализа деревьев синтаксического разбора заголовочных файлов библиотеки.

В качестве дополнительных входных данных инструмент получает списки внутренних функций и типов, проверку которых осуществлять не требуется. В библиотеках на языке Си эти входные данные наиболее актуальны, поскольку отсутствует контроль доступа на уровне компиляции. Выходными данными инструмента является отчёт в формате html с результатами проверки двух версий библиотеки на совместимость. В отчёте все типы проблем совместимости разделены на две группы – проблемы функций и проблемы типов данных. При этом для каждой проблемы, связанной с изменением типа данных, указывается, на какие экспортируемые функции это изменение могло повлиять. Все изменения в отчёте распределяются по трём уровням потенциального риска для пользователей библиотеки. Помимо изменений, которые могут повлиять на несовместимость на бинарном уровне, также

идентифицируются изменения, которые могут повлиять на совместимость на уровне исходных кодов.

3.2. Семантическое тестирование обратной совместимости

Под семантическим тестированием обратной совместимости подразумевается тестирование, которое нацелено на выявление несовместимостей, в том числе, на семантическом уровне, то есть на уровне поведения экспортируемых функций. Наиболее распространённым подходом к семантическому тестированию обратной совместимости является проведение обычного функционального тестирования новой версии библиотеки на соответствие требованиям.

В случае проверки обратной совместимости на бинарном уровне применяется также такой подход, как компиляция функциональных тестов со старой версией библиотеки и запуск этих тестов на новой версии.

Ещё одним логичным шагом при семантическом тестировании обратной совместимости могло бы быть использование старой версии библиотеки в качестве эталона для вынесения вердикта о корректности поведения новой версии библиотеки, но авторам о применении такого подхода на практике ничего не известно. По всей видимости, это обусловлено тем, что в случаях, когда обратной совместимости уделяется серьёзное внимание, также много внимания уделяется и качеству библиотек в целом, а значит и их функциональному тестированию, что позволяет успешно решать и задачу обеспечения обратной совместимости на семантическом уровне.

4. Обнаружение специфических видов ошибок

Ряд ошибок в программах проявляются не сразу и/или опосредованным образом, поэтому традиционные подходы функционального тестирования не эффективны для их выявления. В этом разделе мы рассмотрим два класса таких ошибок в контексте тестирования ОС: утечки ресурсов и гонки по данным. Также в этом разделе будут рассмотрены ошибки, связанные с некорректной обработкой некорректных входных данных.

В качестве интересного факта следует упомянуть, что анализ исправлений ошибок в стабильных версиях ядра ОС Linux за один календарный год [33] показал, что гонки по данным и утечки ресурсов оказались наиболее распространёнными видами исправляемых ошибок: они встречаются в 17% и 16% случаев соответственно.

4.1. Обнаружение утечек ресурсов

Под утечкой ресурсов понимаются ситуации, когда ресурс, выделенный для определённого использования, перестаёт быть нужным, но не освобождается или, другими словами, не возвращается в пул свободных ресурсов. В результате свободные ресурсы могут исчерпаться, и функциональность

системы окажется ограниченной, несмотря на то, что часть «занятых» ресурсов данного вида на самом деле никому не нужна.

Утечки наиболее неприятны для компонентов с продолжительным временем жизни, так как в этом случае они могут накапливаться и привести к нехватке ресурсов. Большинство компонентов ОС обладают длительным временем жизни или, как в случае с системными библиотеками, могут оказаться частью программы с длительным временем жизни. Но наиболее неприятны утечки для ядра ОС, так как время его жизни совпадает со временем непрерывной работы всей системы в целом. Кроме того, в отличие от пользовательских программ, где по окончании работы многие категории используемых программой ресурсов (память, файловые дескрипторы и др.) освобождаются автоматически, ядро ОС должно освобождать все ресурсы явно.

Для пользовательских приложений существует множество инструментов, помогающих обнаруживать утечки, в первую очередь утечки памяти, с помощью специализированных библиотек или более разносторонних подходов, таких как, например, реализованные в инструменте Valgrind [34]. Возможности по обнаружению утечек в ядре ОС более ограничены. Рассмотрим доступные инструменты на примере ядра ОС Linux.

В ядре Linux есть встроенный инструмент kmemleak [35], предназначенный для выявления утечек памяти. Этот инструмент поддерживает список выделенных участков памяти, используя дополнительный код в реализации функций выделения и освобождения памяти. Для выявления утечек памяти используется сканирование всей памяти, используемой ядром, на предмет наличия в них последовательностей байтов, которые могут быть указателем на ту или иную выделенную область памяти. Если при сканировании не обнаружилось возможного указателя на какой-то выделенный участок памяти, то этот участок памяти считается потерянным.

Однако, у такого метода выявления утечек памяти есть следующие недостатки.

- Сканирование всей памяти ядра — долгий процесс. Даже с использованием различных оптимизаций сканирование может выполняться несколько минут.
- Критерий утечек памяти, основанный на поиске возможного указателя на выделенный участок памяти, неточен. Возможны как ложные срабатывания (указатель на выделенную область памяти может хранится в памяти неявно), так и пропущенные ошибки (найденная последовательность байт может, в реальности, и не быть указателем).

Из-за этих недостатков применение kmemleak для автоматических тестов затруднительно, так как поиск утечек памяти будет сильно увеличивать время тестов, а результат должен дополнительно проверяться человеком на предмет

ложных срабатываний. Что до пропущенных ошибок, то их надо выявлять другими инструментами.

На основе платформы KEDR был разработан инструмент KEDR Leak Check, который может использоваться для более эффективного выявления утечек памяти в модулях ядра. Перехват вызовов функций, обеспечиваемый платформой KEDR, в этом инструменте используется для отслеживания выделенных участков памяти. Критерием же отсутствия утечек памяти в варианте использования по умолчанию является отсутствие выделенных участков памяти на момент выгрузки модуля.

KEDR Leak Check решает проблемы скорости и точности, присущие kmemleak. Это позволяет использовать KEDR Leak Check при автоматическом тестировании. Кроме того, данный инструмент выигрывает у kmemleak в плане перечисленных ниже удобств использования и расширяемости.

- Для использования kmemleak ядро должно быть собрано с соответствующей опцией, которая на большинстве дистрибутивов по умолчанию отключена. Для использования KEDR Leak Check пересборка ядра обычно не требуется, так как все необходимые для его работы опции ядра включены по умолчанию.
- KEDR Leak Check выявляет утечки только в одном модуле ядра, не перемешивая их с утечками в других модулях и в основной части ядра.
- Функциональность KEDR Leak Check может быть расширена для выявления утечек других типов ресурсов. Это расширение достигается путём написания дополнительных модулей. Для аналогичного расширения функциональности kmemleak необходимо модифицировать код ядра.

Из ограничений KEDR Leak Check можно отметить следующие.

- Для обнаружения утечек памяти необходима выгрузка модуля. kmemleak же может выявлять утечки даже при работающем драйвере.
- Инструмент не применим для обнаружения утечек в коде основной части ядра.
- Для корректной работы KEDR Leak Check необходим перехват большего количества функций, чем для kmemleak.
- Некоторые ресурсы выделяются основным кодом ядра перед вызовом некоторой функции драйвера (так называемой, callback-функции), а освобождать их должен драйвер. Случается и обратная ситуация, когда ресурс выделяется драйвером, а освобождается после того, как callback-функция возвратит управление основному коду ядра. Для корректного выявления утечек памяти в таких случаях необходимо отслеживать вызов/возврат из соответствующих callback-функций.

4.2. Обнаружение гонок по данным

Гонками по данным обычно называют ситуации, при которых два параллельно работающих потока управления «одновременно» обращаются к разделяемым данным и получают при этом некорректный результат. Например, если одна функции будет читать значение составной переменной из памяти, а другая будет записывать туда новое значение, то может оказаться, что читатель прочитает часть старого значение, после чего значение переменной обновится писателем, и вторую часть переменной читатель прочитает уже из её нового значения. В результате у читателя может оказаться значение, которое не соответствует ни старому значению переменной, ни новому. Аналогичная ситуация может сложиться и при наличии двух одновременно работающих писателей. В этом случае неконсистентное значение может оказаться записанным в переменную.

Гонки по данным могут происходить и на нескольких разделяемых переменных, связанных между собой семантическими связями. Как правило, такие гонки называются высокоуровневыми. Ещё одним типичным примером гонок является ситуация, когда одна функция работает с указателем на динамическую память, а параллельно работающая функция эту память освобождает в середине работы первой функции, после чего последующие разыменования указателя в ней могут привести к непредсказуемым последствиям.

Поскольку гонки случаются непредсказуемым образом при наложении определённых событий во времени, а последствия могут проявляться не сразу, да к тому же в коде, никак не связанном с тем кодом, где присутствует ошибка, то искать такие ошибки очень непросто, и известны случаи, когда корень таких ошибок с весьма неприятными последствиями удавалось найти лишь спустя десятки месяцев достаточно напряжённых поисков.

Для поиска гонок в пользовательских приложениях существует некоторое количество доступных инструментов, таких как Helgrind [36] и Google ThreadSanitizer [37]. Мы рассмотрим более детально, какие техники могут применяться для ядра ОС.

4.2.1 Kernel Strider и ОС2000 Data Race Detector

И Helgrind, и ThreadSanitizer построены на схожих принципах. В инструментах существует часть, ответственная за сбор информации обо всех доступах анализируемой программы к памяти и обо всех обращениях к примитивам синхронизации, и вторая часть, которая анализирует собранную информацию и сообщает о подозрительных местах. Helgrind и ThreadSanitizer первой версии собирают информацию при помощи инфраструктуры платформы Valgrind, а ThreadSanitizer второй версии — за счёт инструментации кода, выполняемой в ходе компиляции программы. Поскольку Valgrind в принципе не совместим с пространством ядра, то Google

финансировал[1] совместный с ИСП РАН проект Kernel Strider [38] по разработке компонента, собирающего информацию для ThreadSanitizer, на основе платформы KEDR. Следует отметить, что инструментация кода в компиляторе, которая используется в ThreadSanitizer второй версии, может быть адаптирована для применения в пространстве ядра, но на момент старта проекта второй версии инструмента ThreadSanitizer ещё не существовало, и по состоянию на текущий момент такой адаптации пока не реализовано.

Для реализации Kernel Strider потребовалось значительно расширить возможности платформы KEDR с целью перехвата не только вызовов функций, но и выполнения машинных операций, работающих с оперативной памятью. Целью перехвата является трасса обращений к ячейкам оперативной памяти и вызовов примитивов синхронизации, которая поступает для дальнейшего анализа на наличие состояний гонок с помощью Google ThreadSanitizer.

Использование Kernel Strider в его базовой конфигурации возможно лишь в «ручном» режиме, поскольку без дополнительной настройки на тестируемый модуль инструмент выдаёт много ложных срабатываний. Причина большинства из этих ложных срабатываний в том, что на порядок вызова callback-функций модуля ядро накладывает дополнительные ограничения, которые никак не учитываются инструментом. Один из способов настройки инструмента — аннотация тестируемого модуля. Аннотации могут описывать дополнительные отношения синхронизации, которые определяются устройством конкретного модуля.

Другой способ настройки инструмента — использование Kernel Strider как платформы для перехвата вызовов функций по аналогии с KEDR. С помощью такого перехвата можно добавлять в трассу события синхронизации, эквивалентные ограничениям, накладываемым ядром на вызов callback-функций драйвера. Такой способ настройки сложнее, но более универсальный — получившаяся настройка может использоваться не только для конкретного драйвера, а для целого семейства драйверов. В данный момент идёт настройка инструмента на семейства драйверов файловых систем и сетевых карт.

Для обнаружения гонок в ядре ОСРВ применялся схожий подход, реализованный в инструменте ОС2000 Data Race Detector. Этот инструмент представляет собой специально модифицированный эмулятор центрального процессора, который в ходе своей работы собирает информацию обо всех операциях с памятью и обо всех вызываемых примитивах синхронизации. Дальнейшая обработка собранной информации также выполняется при помощи Google ThreadSanitizer.

[1] Google Research Award 2011 "Instrumentation and Data Collection Framework for Dynamic Data Race Detection in Linux Kernel Modules".

4.2.2 Race Hound

Инструмент Kernel Strider довольно сложен в реализации и требует настройки на каждое семейство драйверов. Взамен он позволяет выявлять состояния гонки, которые не произошли при данном выполнении кода драйвера, но могли бы произойти.

Инструмент Race Hound [39], реализующий идеи, предложенные в инструменте DataCollider для платформы Windows, и также разработанный в ИСП РАН, позволяет модифицировать последовательность выполнения кода и даёт шанс гонке реально произойти. Модификация времени исполнения кода осуществляется за счёт вставки дополнительных циклов ожидания после тех или иных инструкций кода. Помимо вставки дополнительных циклов ожидания, выполняется наблюдение за обращениями к участкам памяти, к которым обращается инструкция кода. В случае, если это наблюдение выявило ситуацию, подходящую под определение состояния гонки, об этом факте сообщается пользователю.

Для вставки кода после инструкций кода используется существующий механизм Kernel Probes, для наблюдения за обращениями к участкам памяти используются аппаратные точки прерывания. Количество аппаратных точек прерывания определяется архитектурой операционной системы, и их количество обычно невелико (например, x86 и x86_64 поддерживают до четырёх аппаратных точек прерывания), поэтому инструмент Race Hound может одновременно наблюдать только за ограниченным числом ячеек памяти. Для слежения за бо́льшим количеством ячеек их набор меняется с течением времени по рандомизированному алгоритму.

Такие особенности реализации делают процесс обнаружения гонок с помощью Race Hound вероятностным: чем чаще повторяется тот или иной участок кода, тем больше вероятность воспроизвести в нем состояние гонки, если, конечно, оно есть. При этом гарантированно найти все состояния гонки невозможно.

Тем не менее, Race Hound является хорошим инструментом для доказательства возможного состояния гонки, обнаруженного другим инструментом, например, Kernel Strider. В этом случае Race Hound наблюдает только за заранее заданными ячейками и вставляет дополнительные циклы ожидания только после заранее заданных инструкций. С помощью такого совместного использования Kernel Strider и Race Hound было обнаружено и доказано 3 состояния гонки в сетевых драйверах ОС Linux.

4.3. Тестирование устойчивости

Ещё одним видом специфических ошибок является некорректная обработка ОС некорректных входных данных. Этот вид ошибок весьма важен именно для ОС, поскольку к ним предъявляются повышенные требования по

устойчивости к атакам и вредоносным действиям со стороны недоверенных приложений, сетевых контрагентов, подключаемых внешних устройств и т. д.

К тестированию устойчивости существуют различные подходы. Один из подходов — рассматривать требования устойчивости как частный случай функциональных требований и, соответственно, проводить тестирование устойчивости в рамках функционального тестирования. Однако, это далеко не всегда удобно, т. к. может оказаться, что функциональные тесты, нацеленные на работу с корректными аргументами, и тесты устойчивости, нацеленные на передачу всевозможных некорректных данных, удобнее строить на основе различных подходов.

Другой подход заключается в развитии генераторов тестов работоспособности с целью поддержки перебора всевозможных некорректных данных по отдельным аргументам при условии, что остальные данные являются корректными (часто такой подход называют fuzz testing или fuzzing). Опыт генерации тестов с чисто случайными данными показывает малую эффективность, ввиду того, что большинство сгенерированных векторов входных данных не проходят далее первых тривиальных проверок в начале функции и, соответственно, просто не добираются до последующего кода, содержащего ошибки. Поэтому генерация корректных значений для части аргументов играет очень важную роль. Для этого обычно применяются аннотации, схожие с описанными выше генераторами тестов работоспособности. Наиболее успешным представителем инструментов данного класса является инструмент Trinity [40], предназначенный для тестирования устойчивости системных вызовов ОС Linux.

Дальнейшим развитием автоматического подхода являются инструменты, реализующие идею Concolic (или DART) тестирования. Идея заключается в одновременном тестировании целевого компонента и символическом выполнении его кода с целью выявления точек ветвления в программе и автоматическим подбором входных данных, которые бы позволили свернуть в этой точке ветвления не туда, куда пошло текущее выполнение, а в другую сторону. Данный подход является достаточно популярным направлением исследований в настоящее время и существует множество инструментов, его реализующих. В ходе исследования этого направления в ИСП РАН был разработан инструмент Avalanche [41], который продемонстрировал способность находить ошибки в коде, работающем в пространстве пользователя. Кроме того, в ИСП РАН ведутся работы по исследованию эффективности применения методов Concolic тестирования к коду ядра ОС Linux на основе адаптации инструментов S2E [42]. В качестве целевого компонента для тестирования используются драйвера файловых систем.

5. Тестирование производительности ОС

Тестирование производительности в инженерии программного обеспечения определяется как тестирование, которое проводится с целью определения, как

быстро работает вычислительная система или её часть под определённой нагрузкой. Такое тестирование может также служить для проверки и подтверждения других атрибутов качества системы, таких как масштабируемость, надёжность и потребление ресурсов. Заметим, что в отличии от других видов тестирования, тестирование производительности обычно не ставит своей целью выявление нарушений заранее зафиксированных требований. Когда же речь идёт о производительности, такие требования зачастую отсутствуют. Это вызвано тем, что конкретные значения разнообразных показателей производительности для одной и той же программной системы сильно зависят от возможностей аппаратных платформ, на которых может выполняться эта программная система. Поэтому постановка задачи тестирования производительности сложных программных систем, к которым относятся и операционные системы, требует дополнительной конкретизации. Набор задач тестирования определяется как архитектурой программной системы и ее аппаратной платформы, так и целями тестирования. В достаточно общем случае к задачам тестирования производительности относится:

1. определение конкретных показателей производительности, свойственных каждой конкретной программной системе или классу систем;
2. определение зависимости показателей производительности программной системы от конфигурации системы и характеристик программно-аппаратной платформы, на которой она выполняется;
3. для каждого из показателей определение такой методики измерения значений этого показателя, которая обладает известной точностью (погрешностью) измерений;
4. создание программных средств для измерения значений показателей производительности по определённым методикам и для анализа контролируемых зависимостей между показателями производительности и влияющими на них факторами;
5. контроль изменений показателей производительности в процессе развития программной системы.

Таким образом, если разработчик программной системы предоставляет данные о значениях показателей производительности, которые должны соблюдаться в этой системе при определённых условиях, то тестирование производительности заключается в проверке соответствия измеренных значений заявленным. В противном случае тестирование производительности заключается в контроле изменений значений показателей с целью своевременного выявления случаев деградации работоспособности системы и условий, при которых такая деградация стала возможной.

Применительно к ОС и, в особенности, к ОС жёсткого реального времени задача тестирования производительности имеет особое значение. Это значение определяется ролью операционной системы в обеспечении

функционирования современных вычислительных компьютерных систем и влиянием производительности ОС не только на производительность, но и на функциональные характеристики всей системы в целом. В отличие от универсальных ОС основной задачей ОСРВ является своевременность (timeliness) выполнения обработки данных. Поэтому в качестве основного требования к ОСРВ выдвигается требование обеспечения предсказуемости (детерминированности) поведения системы в наихудших внешних условиях. Тестирование производительности помогает проверить выполнение этого требования.

5.1. Показатели производительности

Вообще говоря, номенклатура показателей производительности ОС зависит от её назначения и от стандарта (например, POSIX, ARINC-653 и т. п.), на основе которого разработана конкретная ОС и определяющего состав системных ресурсов и сервисов. Тем не менее можно сформулировать ряд низкоуровневых показателей, которые будут применимы к подавляющему большинству ОС. К таким показателям относятся:

- время обработки прерывания;
- время создания потока управления;
- время переключения между потоками управления (переключение контекстов потоков);
- время переключения процессов (переключение контекстов процессов);
- время создания ресурсов (семафоров, очередей, мьютексов, таймеров и т. п.);
- пропускная способность системы обмена сообщениями между потоками управления и/или процессами.

Этот список можно дополнить временными показателями выполнения каждого системного сервиса, в совокупности покрывающих все отдельные базовые операции, предоставляемые операционной системой приложениям. Полученная в результате детальная номенклатура показателей позволяет оценить производительность всех основных операций и сервисов ОС, которые, собственно, и влияют на работоспособность приложений.

5.2. Способы измерений показателей производительности

Основными способами измерения показателей производительности можно считать следующие:

- использование специальной внешней измерительной аппаратуры для считывания сигналов с выделенных для целей измерений портов целевого модуля, работающего под управлением целевой ОС;
- инструментирование исходного кода ОС путём вставки команд считывания текущего времени в начале и в конце участка кода, выполняющего измеряемое действие;
- создание специальных приложений, которые вызовами интерфейсных системных сервисов заставляют тестируемую ОС выполнять (возможно, многократно) измеряемое действие.

Каждый из перечисленных способов обладает своими достоинствами и недостатками. Использование измерительной аппаратуры и инструментирование кода ОС обеспечивают самое точное измерение, т. к. позволяет исключить из последовательности действий ОС всё то, что не относится к измеряемому показателю. Однако выполнение дополнительных команд будет неизбежно вносить возмущения в работу компонентов ОС, что особенно недопустимо в случае ОСРВ. Использование внешней измерительной аппаратуры требует незначительного инструментирования кода ОС, однако возможности этого способа ограничены аппаратными возможностями целевого модуля. Кроме того, исходный код коммерческих ОС обычно недоступен. Поэтому мы считаем, что для решения задачи тестирования производительности ОС надо сосредоточиться на разработке специальных приложений-измерителей. Такой подход позволяет оценить показатели производительности «с точки зрения приложений». По этому пути идут исследователи и разработчики программных средств измерения показателей реального времени для специализированных окружений реального времени ядра ОС Linux (таких как RTAI, Xenomai) [43-47]. Главная задача при таком подходе — обеспечение предсказуемой точности измерений. Для решения этой задачи требуется разработать методики измерения каждого отдельного показателя. Эти методики зависят не только от содержания каждого показателя, но и от состава и функциональности интерфейсных сервисов ОС.

5.3. Программные средства измерения показателей производительности

В ИСП РАН разработан прототип инструмента, измеряющего показатели производительности ОСРВ, поддерживающих стандарт ARINC-653. Список показателей производительности сформирован на основе выделения ключевых функциональных блоков ARINC-653 и включает в себя:

- накладные расходы ОС, связанные со временем переключения между ARINC-разделами (процессами);
- накладные расходы ОС, связанные со временем переключения между ARINC-процессами (потоками управления);
- пропускная способность потоковых (queuing) каналов данных между ARINC-разделами;
- пропускная способность перезаписываемых (sampling) каналов данных между ARINC-разделами;
- пропускная способность потоковых (buffer) каналов данных между

ARINC-процессами;
- пропускная способность перезаписываемых (blackboard) каналов данных между ARINC-процессами.

Разработанный инструмент создаёт сценарии использования ресурсов ОС и производит все измерения, используя только сервисы ОСРВ, предусмотренные стандартом ARINC-653. Таким способом эмулируется работа пользовательского приложения под управлением тестируемой ОСРВ и исключается вмешательство измерителя в работу компонентов ОС. Разработанный измерительный инструмент апробирован на ОСРВ ОС3000/ОС4000 (разработка НИИСИ РАН) и WindRiver VxWorks-653 и может быть использован для сравнительной оценки характеристик производительности разных ОСРВ на одной и той же аппаратной платформе.

6. Комплексный подход к тестированию ОС

6.1. Конфигурационное тестирование ОС

Существует ещё одна особенность ОС, которая значительно влияет на вопросы организации тестирования. А именно, ОС, как правило, обладают огромным набором конфигурационных параметров, регулирующих особенности поведения её компонентов, а также ОС предназначены для работы на разнообразном оборудовании, которое в свою очередь имеет многочисленные конфигурации. В результате возникает бесчисленное множество комбинаций, в которых может работать ОС, и встаёт вопрос о том, как можно протестировать эти комбинации адекватным образом.

Понятный ответ существует для ОС, применяемых в ответственных системах, от корректности поведения которых может зависеть жизни людей. В таких системах существует однозначно выбранная конфигурация оборудования и конфигурация ОС, которая и подлежит тщательному тестированию.

В случае ОС общего назначения разумным подходом является выделение наиболее важных конфигурационных опций и построение для них покрывающих множеств [48], чтобы сформировать ограниченное число комбинаций, в которых встретятся хотя бы все возможные пары значений каждой из пар опций. Но и число таких комбинаций всё равно оказывается достаточно велико, чтобы проводить полномасштабное тестирование для каждой из них. Тем не менее, упрощённое тестирование каждой комбинации имеет смысл провести. В качестве минимального варианта могут выступать простейшие тесты, подтверждающие возможность скомпилироваться и загрузиться в соответствующей комбинации. Для более детального тестирования возможно провести ручной отбор комбинаций, например, на основе дополнительных соображений о наиболее востребованных конфигурациях. Причём для каждого вида тестирования набор целевых конфигураций может подбираться индивидуально.

6.2. Комплексная система управления тестированием

В предыдущих разделах мы идентифицировали множество задач, возникающих при системном подходе к тестированию ОС, в том числе:

- идентификация компонентов ОС и предоставляемой ими функциональности;
- классификация компонентов ОС по их важности с целью определения целевого уровня качества их функционального тестирования:
  - тщательное тестирование;
  - среднее тестирование;
  - тестирование работоспособности;
  - отсутствие тестирования;
- выделение компонентов, для которых требуется проведение дополнительных тестов для выявления специфических видов ошибок;
- идентификация компонентов и их интерфейсов, к которым предъявляются требования по устойчивости к атакам и вредоносным действиям их контрагентов;
- определение требований к обеспечению обратной совместимости ОС и компонентов, для которых требуется проведение соответствующего тестирования;
- определение целевых показателей производительности ОС и методик их измерения;
- проектирование и разработка тестовых наборов и средств измерения показателей производительности в соответствии с целями, сформированными согласно вышеизложенным пунктам;
- выбор целевых комбинаций конфигураций ОС и аппаратного обеспечения для каждого из тестовых наборов.

Следующая задача, которая становится актуальной после появления работающих тестовых наборов, заключается в построении процесса выполнения тестов для каждой новой версии ОС. Наш опыт показывает, что с самого начала необходимо планировать разработку комплексной системы управления тестированием, которая будет решать следующие задачи:

- управление всеми тестовыми стендами (как реальными, так и виртуальными) и другими отведёнными аппаратными ресурсами;
- управление автоматическим выполнением тестовых наборов;
- сбор, обработка и визуализация информации о результатах тестирования;
- сравнительный анализ результатов тестирования по версиям ОС, по аппаратным платформам, по конфигурациям и т. д.;
- управление версиями ОС и тестовых наборов.

Первый опыт разработки такой среды в ИСП РАН воплотился в инструменте Linux Distribution Checker [49], который предназначен для проведения

тестирования дистрибутивов ОС Linux на соответствие требованиям стандарта LSB. Вследствие того, что основными объектами тестирования являлись системные библиотеки ОС и того, что инструмент предназначен для проведения тестирования в рамках сертификационной программы LSB, задачи управления тестовыми стендами и версиями ОС были не актуальными и не реализованы в Linux Distribution Checker. Более полный набор функциональности был реализован в специализированной системе управления лабораторией тестирования ОСРВ, которая включает в себя и поддержку управления аппаратными ресурсами, в том числе для проведения распределённого тестирования на нескольких машинах одновременно, и автоматическую установку новых версий ОС, и настройку аппаратуры тестовых стендов под целевую конфигурацию тестирования, и генерацию сравнительных отчётов, и визуализацию данных об измерении производительности ОС.

7. Заключение

Подводя итог, следует сделать вывод, что для решения задачи тестирования ОС в целом необходимо умело сочетать разные методы построения тестов, учитывая особенности объекта тестирования, возможности доступных техник и инструментов, а также аккуратно формируя планы в соответствии с целями проекта, имеющимися ресурсами и временны*ы*ми ограничениями.

В качестве ориентира можно использовать следующие усреднённые оценки трудоёмкости разработки тестов для одной условной интерфейсной функции целевого компонента. Тесты работоспособности при поддержки инструментами типа Azov или API Sanity Autotest могут разрабатываться со скоростью, позволяющей покрывать порядка 100 целевых функций в день усилиями одного инженера. Эта оценка дана в предположении однородности целевых функций. Если рассматривать целевые функции из различных малопохожих библиотек, то следует ожидать значительного падения производительности. Тесты среднего уровня качества с ожидаемым уровнем покрытия 70-80% строк исходного кода могут разрабатываться со скоростью порядка 2 функций в день. Разработка тестов высокого качества занимает порядка 5 дней на один аспект функциональности целевого компонента, что, по нашему опыту, в среднем соответствует одной функции в два дня. Также следует отметить, что разработка тестов высокого качества возможна лишь при наличии достаточно полной документации, иначе её приходится восстанавливать в ходе разработки тестов.

Дальнейшее развитие исследований в области тестирования ОС в ИСП РАН планируется по следующим направлениям:

- совмещение статических методов анализа программ и динамического тестирования, в частности:
  - исследование эффективности применения методов Concolic тестирования к коду ядра ОС Linux на основе развития инструментов S2E, в первую очередь, на примере драйверов файловых систем;
  - поиск потенциальных гонок по данным при помощи методов статического анализа и целенаправленная проверка выдвинутых гипотез динамическими методами, такими как RaceHound;
- сравнительный анализ стратегий систематической симуляции сбоев на примере драйверов файловых систем ОС Linux;
- развитие методик и инструментов измерений производительности ключевых компонентов ОС;
- применение методов дедуктивной верификации для доказательства корректности наиболее ответственных компонентов ОС.

8. Благодарности

Авторы выражают признательность К. Власову, Р. Зыбину, А. Пономаренко, В. Рубанову, Е. Чернову и Е. Шатохину за активное участие в ряде описанных проектов. На работы по тестированию ОСРВ значительное влияние оказали соображения и советы сотрудников НИИСИ РАН и, в первую очередь, главного конструктора ОС2000/3000/4000 А.Н. Годунова. В разработке системы тестирования файловых систем ОС Linux активно участвовали сотрудники Лаборатории системного программирования Российско-Армянского Славянского университета. Кроме того, хотя описанные работы базируются на технологиях, созданных в ИСП РАН, многие работы ведутся как открытые проекты, и в развитие инструментов тестирования ОС вносят свой вклад не только сотрудники ИСП РАН, но другие участники этих проектов. Авторы также выражают им свою глубокую признательность.

Список литературы

[1] В.В. Кулямин, Н.В. Пакулин, О.Л. Петренко, А.А. Сортов, А.В. Хорошилов. Формализация требований на практике. Препринт №13. М.: ИСП РАН, 2006.

[2] В.В. Кулямин, А.К. Петренко, В.В. Рубанов, А.В. Хорошилов. Формализация интерфейсных стандартов и автоматическое построение тестов соответствия. «Информационные технологии», №8, 2007, С.1-8.

[3] М.В. Екимов, И.В. Ковернинский, А.В. Хорошилов. “АРМ ПТ: создание системы проектирования тестов для критических систем на основе СПО”. Сборник докладов Всероссийской конференции “Свободное программное обеспечение – 2010”, Санкт-Петербург, 26-27 октября 2010 г.

[4] Сайт инструмента Requality, http://forge.ispras.ru/projects/reqdb.

[5] Д.Ю. Кичигин. Метод редукции тестового набора для регрессионного интеграционного тестирования. «Программирование», №5, 2009, С.57-69.

[6] Alexey Khoroshilov, Vladimir Rubanov, Eugene Shatokhin. «Automated Formal Testing of C API Using T2C Framework». In Proceedings of the Third International Symposium «Leveraging Applications of Formal Methods, Verification and Validation» (ISoLA 2008), Porto Sani, Greece, October 13-15, 2008. pp.56-70. ISBN 978-3-540-88478-1.

[7] TETWare User Guide, http://tetworks.opengroup.org/documents/3.7/uguide.pdf.
[8] ISO/IEC 23360-1-8:2005, Linux Standard Base (LSB) Core Specification 3.1. Geneve: ISO, 2005.
[9] А.В. Хорошилов. Linux Standard Base: история успеха?. Труды Института Системного Программирования РАН, Том 10, 2006. С.29-50.
[10] Список ошибок, обнаруженных в ходе тестирования библиотек ОС Linux. http://linuxtesting.org/results/impl_reports.
[11] Н.В.Пакулин, А.В.Хорошилов. Разработка формальных моделей и тестирование соответствия для систем с асинхронными интерфейсами и телекоммуникационных протоколов. Программирование, №6, 2007, С.26-55.
[12] А.В. Баранцев, И.Б. Бурдонов, А.В. Демаков, С.В. Зеленов, А.С. Косачев, В.В. Кулямин, В.А. Омельченко, Н.В. Пакулин, А.К. Петренко, А.В. Хорошилов. “Подход UniTesK к разработке тестов: достижения и перспективы”. Труды Института системного программирования РАН, №5, 2004. С.121-156.
[13] А.В.Хорошилов. Спецификация и тестирование систем с асинхронным интерфейсом. Препринт №12. М.: ИСП РАН, 2006.
[14] IEEE 1003.1-2008. Information Technology — Portable Operating System Interface (POSIX). New York: IEEE, 2008.
[15] ARINC. ARINC Specification 653P1-3: Avionics Application Software Standard Interface Part 1 - Required Services. Aeronautical Radio INC, Maryland, USA, 2010.
[16] Alexey Grinevich, Alexey Khoroshilov, Victor Kuliamin, Denis Markovtsev, Alexander Petrenko, Vladimir Rubanov. “Formal Methods in Industrial Software Standards Enforcement”, PSI 2006, LNCS Vol. 4378, 2006, pp. 446-455.
[17] A.Maksimov. Requirements-based conformance testing of ARINC 653 real-time operating systems // Proceedings of the Data Systems In Aerospace (DASIA 2010) conference, 2010, ESA SP-682.
[18] Описание возможностей Fault Injection. https://www.kernel.org/doc/Documentation/fault-injection/fault-injection.txt.
[19] Eugene Shatokhin. Using Dynamic Analysis To Hunt Down Problems in Kernel Modules. Presentation at LinuxCon Europe 2011, Czech Republic, Prague, 26-28 October 2011.
[20] Сайт платформы KEDR. http://linuxtesting.org/kedr.
[21] Проект по верификации модулей драйверов файловых систем. http://linuxtesting.org/spruce.
[22] V. Kuliamin. Standardization and Testing of Mathematical Functions. // Proc. of PSI’2009, Novosibirsk, Russia, June 2009, LNCS 5947. pp. 257-268, Springer, 2009.
[23] Р.С. Зыбин, В.В. Кулямин, А.В. Пономаренко, В.В. Рубанов, Е.С. Чернов. Автоматизация массового создания тестов работоспособности //Программирование, 34(6):64-80, 2008.
[24] А.В. Пономаренко, В.В. Рубанов, А.В. Хорошилов. Автоматическая генерация тестов для C/C++ библиотек. Сборник докладов Седьмой конференции разработчиков свободных программ, Переславль, 26-27 июля 2010 г.
[25] Сайт API Sanity Autotest. http://forge.ispras.ru/projects/api-sanity-autotest.
[26] Инструмент LSB libchk. http://bzr.linuxfoundation.org/loggerhead/lsb/devel/misc-test/files.
[27] Описание Android Compatibility Test Suite. http://source.android.com/compatibility.
[28] chkshlib, http://osr507doc.sco.com/en/man/html.CP/chkshlib.CP.html.
[29] cmpdylib, http://www.opensource.apple.com/source/cctools/cctools-795/man/cmpdylib.1.

[30] cmpshlib, sys-admin.net/ebooks/unix3/mac/ch07_01.htm.
[31] dpkg-gensymbols, http://man.he.net/man1/dpkg-gensymbols.
[32] А. Пономаренко, В. Рубанов, А. Хорошилов. “Система анализа обратной бинарной совместимости библиотек Linux”. Сборник докладов международной конференции “Software Engineering Conference (Russia)”, SEC(R)-2009, сс. 25-31, Москва, 28-29 октября 2009 г.
[33] В.С. Мутилин, Е.М. Новиков, А.В. Хорошилов. “Анализ типовых ошибок в драйверах операционной системы Linux”. Труды Института Системного Программирования, Том 22, 2012, с. 349-374.
[34] Сайт инструмента Valgrind. valgrind.org.
[35] Описание возможностей Kernel Memory Leak Detector. https://www.kernel.org/doc/Documentation/kmemleak.txt.
[36] Описание возможностей Helgrind. http://valgrind.org/docs/manual/hg-manual.html.
[37] Konstantin Serebryany, Timur Iskhodzhanov. ThreadSanitizer: data race detection in practice. In Proceedings of the Workshop on Binary Instrumentation and Applications (WBIA '09). ACM, New York, NY, USA, pp.62-71.
[38] Сайт проекта Kernel Strider. https://code.google.com/p/kernel-strider/.
[39] Сайт проекта Race Hound. http://forge.ispras.ru/projects/race-hound.
[40] Сайт проекта Trinity. http://codemonkey.org.uk/projects/trinity/.
[41] И.К. Исаев, Д.В. Сидоров, А.Ю. Герасимов, М.К. Ермаков. Avalanche: Применение динамического анализа для автоматического обнаружения ошибок в программах использующих сетевые сокеты. Труды Института Системного Программирования, Том 21, 2011, с. 55-70.
[42] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. The S2E Platform: Design, Implementation, and Applications. ACM Trans. Comput. Syst. 30, 1, Article 2 (February 2012), pp.1-49.
[43] A. Barbalace, A. Lunchetta, G. Manduchi, M. Moro, A. Soppelsa and C. Taliercio, “Performance Comparison of VxWorks, Linux, RTAI and XENOMAI in a Hard Real-time Application”, Proc. of Real-Time Conference 2007 15th IEEE-NPSS, (2007), pp. 1-5.
[44] M. Franke, “A Quantitative Comparison of Realtime Linux Solutions”, Chemnitz University of Technology, (2007).
[45] M. D. Marieska, A. I. Kistijantoro and M. Subair, “Analysis and Benchmarking Performance of Real Time Patch Linux and Xenomai in Serving a Real Time Application”, Proc. of International Conf. on Electrical Engineering and Informatics, (2011), pp. 1-6.
[46] J. H. Koh and B. W. Choi, “Performance Evaluation of Real-time Mechanisms for Real-time Embedded Linux”, J. of Institute of Control, Robotics and Systems (in Korean), vol. 18, no. 4, (2012), pp. 337-342.
[47] J. H. Koh and B. W. Choi, “Real-time Performance of Real-time Mechanisms for RTAI and Xenomai in Various Running Conditions”, International Journal of Control and Automation, Vol. 6, No. 1, February, 2013, pp. 235-246.
[48] В.В. Кулямин. Комбинаторная генерация программных конфигураций ОС. Труды Института Системного Программирования, Том 23, 2012, с. 359-370.
[49] Vladimir Rubanov, Denis Silakov. Certification Infrastructure for the Linux Standard Base (LSB). //Proceedings of the second International Workshop on Foundations and Techniques for Open Source Software Certification (OpenCert 2008). Milan, Italy, 2008. pp. 79-88.

Testing of Operating Systems

Gerlits E.A., Kuliamin V.V., Maksimov A.V., Petrenko A.K., Khoroshilov A.V., Tsyvarev A.V.
{gerlits, kuliamin, andrew, petrenko, khoroshilov, tsyvarev} @ispras.ru

Abstract. An operating system is a base stone of any computer system. Failures and bugs in operating system impact the functionality of the system as a whole, that is why correctness and reliability of operating systems are so important. A variety of circumstances make verification and testing of operating systems a complicated issue. The list includes high dependence of operating systems on hardware, their massive internal concurrency, huge number of configuration options, required tolerance to aggressive actions of counteragents and hardware faults, a need for long continuous work without reboot, etc. The paper discusses influence of all the circumstances on testing, describes testing tools and techniques developed in ISPRAS and presents our experience of testing of various components of Linux as well as a few other real-time operating systems.

Keywords: operating system, model-based testing, functional testing, robustness testing. performance testing.