Построение надежных операционных систем, допускающих наличие ненадежных драйверов устройств

Йоррит Хердер (Jorrit N. Herder), Херберт Бос (Herbert Bos), Эндрью Таненбаум (Andrew S. Tanenbaum)

Перевод - Сергей Кузнецов

По материалам сайта: http://www.citforum.ru/

Оригинал: A Lightweight Method for Building Reliable Operating Systems Despite Unreliable Device Drivers (http://www.minix3.org/doc/reliable-os.pdf), Technical Report IR-CS-018, January 2006

Я некоторое время колебался, стоит ли переводить этот материал после публикации своего (очень подробного) пересказа замечательной, на мой взгляд, статьи "Можем ли мы сделать операционные системы надежными и безопасными". В данном случае мы имеем дело не со статьей, а с техническим отчетом, и, конечно, текст является менее качественным (в частности, имеются повторы). Но, подумав, я решил, что высокие технические качества статьи перевешивают ее небольшие литературные недостатки. Во-первых, здесь изложены все основные технические идеи ОС MINIX 3. Их простота и эффективность воодушевляют. Во-вторых, в отчете содержится очень хороший обзор литературы по современным операционным системам. Мне кажется, что этот материал может быть полезен всем людям, интересующимся технологией операционных систем и, прежде всего, преподавателям и студентам ВУЗов. Я проявил дополнительную заботу о читателях и нашел в Internet свободные ссылки на полные тексты большинства статей, перечисленных в списке литературы. Эти ссылки добавлены мною в список литературы. К сожалению, остальные тексты доступны только подписчикам электронных библиотек ACM и IEEE Computer Society.

Сергей Кузнецов

Аннотация

Хорошо известно, что в большинстве случаев аварийные отказы операционных систем происходят из-за ошибок в драйверах устройств. Поскольку драйверы обычно подключаются к адресному пространству ядра, драйвер, содержащий ошибки, может затереть таблицы ядра и привести к аварийному отказу или останову системы. Нам удалось значительно смягчить эту проблему засчет сокращения размеров ядра до абсолютного минимума и выполнения каждого драйвера в виде отдельного непривилегированного процесса в пользовательском адресном пространстве. Кроме того, мы реализовали POSIX-совместимую операционную систему в виде набора процессов, выполняемых в пользовательском режиме. В режиме ядра выполняется только крошечное ядро, состоящее менее чем из 3800 строк исполняемого кода, которое выполняет начальную обработку прерываний, запускает и останавливает процессы и обеспечивает IPC. За счет перемещения практически всей операционной системы в несколько защищенных процессов, выполняемых в пользовательском режиме, мы уменьшили последствия сбоев, поскольку сбой драйвера больше не является фатальным и не приводит к потребности перезагрузки системы. В действительности, в состав нашей системы входит *сервер реинкарнации*, который разработан для борьбы с такими ошибками и часто обеспечивает полное восстановление, прозрачное для приложения и обеспечивающее отсутствие потери данных. Для достижения максимальной надежности в своей разработке мы руководствовались принципами простоты, модульности, наименьшей авторизации и отказоустойчивости. В этой статье обсуждается наш облегченный подход, и приводятся данные о его эффективности и надежности. Кроме того, наша разработка сравнивается с другими подходами к защите драйверов с

использованием обертывания ядра и виртуальных машин.

Совершенство достигается не тогда,
когда уже нечего прибавить,
а когда уже ничего нельзя отнять
Антуан де Сент-Экзюпери. Планета людей [9]

1. Введение

Наиболее острой проблемой многих пользователей является ненадежность компьютеров.

Исследователи в области компьютерной науки привыкли к регулярным сбоям компьютеров и к необходимости через каждые несколько месяцев устанавливать патчи программного обеспечения. Однако подавляющее большинство пользователей считает это отсутствие надежности неприемлемым. Их внутренняя модель работы электронного устройства основывается на опыте использования телевизоров и видеомагнитофонов: вы покупаете устройство, подключаете его к сети, и оно безупречно работает в течение 10 лет. Никаких отказов, никаких регулярных обновлений программного обеспечения, никаких газетных историй об обнаружении новейших представителей бесконечной череды вирусов. Чтобы сделать компьютерные системы более похожими на телевизоры, мы ставим целью своего исследования совершенствование надежности компьютерных систем, и начинаем с операционных систем.

1.1 Почему у систем случаются отказы?

Основная причина аварийных отказов операционных систем кроется в двух принципиальных дефектах разработки, свойственных всем этим системам: наличие слишком большого числа привилегий и отсутствие адекватной изоляции сбоев. Практически все операционные системы состоят из многочисленных модулей, скомпонованных в одном адресном пространстве и образующих единую бинарную программу, которая выполняется в режиме ядра. Ошибка в любом модуле может легко привести к разрушению структур данных в каком-либо другом, не связанным с ним модуле и к мгновенному выходу системы из строя. Причиной, по которой все модули компонуются в единое адресное пространство без поддержки какой-либо защиты между модулями, является Фаустова сделка разработчиков: улучшенная производительность за цену большего числа отказов системы. Ниже мы оценим стоимость этого компромисса.

Тесно связанный вопрос относится к первопричине аварийных отказов. Ведь если бы каждый модуль был безупречным, то не возникала бы потребность в изоляции сбоев между модулями, поскольку не было бы самих сбоев. Мы утверждаем, что большая часть сбоев возникает из-за ошибок программирования, вследствие чрезмерной сложности и использования чужого кода. Исследования показывают, что в программном обеспечении в среднем содержится от одной до шестнадцати ошибок на тысячу строк кода [27, 22, 2], и что верхняя граница этого диапазона явно занижена, поскольку учитывались только те ошибки, которые, в конце концов, удавалось обнаружить. Очевидным заключением является то, что *в большем объеме кода содержится большее число ошибок*. По мере развития программного обеспечения в каждой его новой версии появляется все больше возможностей (и, соответственно, больший объем кода), и часто новая версия является менее надежной, чем предыдущая. В [22] показано, что число ошибок на тысячу строк кода стремится к стабилизации по мере роста числа выпущенных версий, но асимптотически этот показатель отличается от нуля.

Наличие некоторых из этих ошибок позволяет злоумышленникам применять вирусы и черви для заражения и повреждения системы. Так что некоторые якобы наличествующие проблемы «безопасности» в принципе не имеют ничего общего с нарушениями мер безопасности (например, дефектными криптографическими алгоритмами или неустойчивыми протоколами авторизации), а вызываются всего лишь ошибками в коде программ (например, переполнения буферов позволяют выполнять внедренный код). Когда в этой статье мы говорим о «надежности», мы имеем в виду и то, что часто называют «безопасностью», – неавторизованный доступ вследствие ошибки в коде программы.

Вторая проблема состоит в привнесении в операционную систему чужого кода. Наиболее искушенные пользователи никогда бы не позволили сторонней организации вставить незнакомый код в ядро операционной системы, хотя, когда они покупают новое периферийное

устройство и инсталлируют соответствующий драйвер, они именно это и делают. Драйверы устройств обычно пишутся программистами, работающими на изготовителей периферийных устройств, и контроль качества их продукции обычно ниже, чем у поставщиков операционных систем. В тех случаях, когда драйвер относится к open-source, его часто пишет благонамеренный, но не обязательно опытный доброволец, и контроль качества обеспечивается на еще более низком уровне. Например, в Linux частота появления ошибок в драйверах устройств от трех до семи раз выше, чем в других частях ядра [7]. Даже компания Microsoft, у которой имеются стимулы и ресурсы для применения более плотного контроля качества, не может добиться намного лучших результатов: 85% всех аварийных отказов Windows XP обуславливается наличием ошибок в коде драйверов.

В последнее время появились публикации о родственных работах, посвященных изоляции драйверов устройств с использованием аппаратуры MMU [26] и виртуальных машин [19]. Эти методы концентрируются на решении проблем в унаследованных операционных системах; мы обсудим их в разд. 6. В отличие от этого, при применении нашего подхода надежность достигается путем разработки новой облегченной операционной системы.

1.2 Решение: правильная изоляция сбоев

В течение десятилетий в качестве проверенного метода оперирования кодом, не заслуживающим доверия, использовалось размещение его в отдельном процессе и выполнение в пользовательском режиме. Одним из ключевых наблюдений, полученных в исследовании, которому посвящена эта статья, является то, что мощным средством повышения надежности операционной системы является выполнение каждого драйвера в виде *отдельного* процесса в пользовательском режиме с минимальными требуемыми привилегиями. Таким образом, код, потенциально содержащий ошибки, изолируется, и ошибка, скажем, в драйвере принтера может привести к прекращению печати, но не к записи искаженных данных в какие-либо важные структуры данных ядра и выходу системы из строя.

В этой статье мы проводим тщательное различие между крахом операционной системы, после которого требуется перезагрузка компьютера, и сбоем или отказом сервера или драйвера, после которого в нашей системе перезагрузка не требуется. Во многих случаях дефектный драйвер, выполняемый в пользовательском режиме, может быть удален и заменен без потребности в перезапуске других частей операционной системы, выполняемых в пользовательском режиме.

Мы не рассчитываем на то, что вскоре появится код, свободный от ошибок, а если и появится, то, конечно, не в операционных системах, которые обычно пишутся на C или C++. К сожалению, в программах, написанных на этих языках, интенсивно используются указатели, обильный источник ошибок. Поэтому наш подход основан на идеях модульности и изоляции сбоев. Путем разбиения системы на большое число изолированных модулей, каждый из которых выполняется в отдельном процессе в режиме пользователя, нам удалось сократить часть системы, выполняемую в режиме ядра, до абсолютного минимума и предотвратить распространение сбоев, возникающих в других модулях. Уменьшение размеров ядра значительно сокращает число ошибок, которые оно, вероятно, должно содержать. Малый размер также позволяет понизить уровень сложности ядра и облегчить его понимание, что также способствует надежности. Поэтому мы последовали максиме Сент-Экзюпери и сделали ядро настолько небольшим, насколько это позволяют человеческие возможности: менее 3800 строк кода.

Одно из замечаний, постоянно возникающее по поводу таких разработок минимального ядра, касается замедления работы системы из-за дополнительных переключений контекста и копирования данных, которое требуется для обеспечения коммуникаций различных моделей, выполняемых в пользовательском адресном пространстве. Это опасение, в основном, существует по историческим причинам, и мы утверждаем, что эти причины, большей частью, теперь отсутствуют. Во-первых, результаты новых исследований показывают, что разработка минимального ядра не обязательно наносит ущерб эффективности [3, 23, 15]. Уменьшение

размеров ядра при наличии разумных протоколов взаимодействия серверов помогает ограничить масштабность проблемы эффективности. Во-вторых, значительное возрастание мощности компьютеров в последнее десятилетие существенно ослабляет проблему гарантированной производительности, возникающую при модульной разработке. В третьих, мы полагаем, что наступает время, когда большая часть пользователей с удовольствием пожертвует некоторой эффективностью ради улучшенной надежности.

Подробное обсуждение эффективности нашей системы мы представляем в разд. 5. Однако здесь мы кратко упомянем три предварительных показателя эффективности в поддержку нашего довода о том, что системы с минимальным ядром не обязательно должны быть медленными. Во-первых, измеренное время выполнения простейшего системного вызова getpid составляет 1.01 мсек на процессоре Athlon с частотой 2.2 Ггц. Это означает, что программа, производящая 10000 системных вызовов в секунду, тратит на переключение контекста всего 1% времени ЦП, а 10000 системных вызовов в секунду производят лишь немногие программы. Во-вторых, наша система способна в течение 4 секунд полностью произвести свою компоновку, включая ядро и все части, выполняемые в режиме пользователя (при этом компилируются 123 файла и совершается 11 редактирований связей). В третьих, время начальной загрузки системы с момента выхода из монитора многовариантной загрузки до выдачи приглашения ко входу в систему составляет менее 5 секунд. После этого операционная система, полностью совместимая с POSIX, готова к использованию.

1.3 Вклад этой статьи

Исследование, результаты которого описываются в этой статье, было направлено на выработку ответа на следующий вопрос: как избежать ситуаций, в которых серьезная ошибка в драйвере устройства (например, использование неверного указателя или наличие бесконечного цикла) приводит к аварийному отказу или зависанию всей операционной системы?

Наш подход состоял в разработке надежной мультисерверной операционной системы поверх крошечного ядра, не содержащего какого-либо внешнего, ненадежного кода. Для обеспечения должной изоляции сбоев каждый сервер и драйвер выполняется в пользовательском режиме в рамках отдельного процесса. Кроме того, мы добавили механизмы для восстановления после возникновения распространенных сбоев. Мы подробно описываем средства поддержки надежности и объясняем, почему они отсутствуют в традиционных монолитных операционных системах. Мы также обсуждаем полученные показатели эффективности системы и показываем, что средства поддержки надежности замедляют систему на 5-10%, но делают ее устойчивой к наличию неверных указателей, бесконечных циклов и других ошибок, которые привели бы к аварийному отказу или зависанию традиционных операционных систем.

Хотя ни один из отдельных аспектов нашего подхода (ядра небольшого размера, драйверы устройств, выполняемые в пользовательском режиме, или мультисерверные системы) не является новым, никто раньше не собирал вместе все эти части для построения небольшой, гибкой, модульной UNIX-подобной системы, являющейся гораздо более отказоустойчивой, чем обычные системы семейства UNIX, и теряющий только 5-10% эффективности по сравнению с нашей базовой системой, содержащей драйверы в ядре.

Кроме того, наш подход в корне отличается от других аналогичных работ, поскольку мы *не* фокусируемся на массовых операционных системах. Вместо этого мы получаем надежность на основе новой, облегченной архитектуры. Вместо того чтобы добавлять вспомогательный код, повышающий надежность ненадежных систем, мы расщепляем операционную систему на небольшие компоненты и достигаем надежности за счет модульности системы. Хотя наши методы неприменимы к унаследованным операционным системам, мы надеемся, что они помогут сделать более надежными будущие операционные системы.

Мы начинаем статью со сравнения нашей разработки со структурами других операционных

систем (разд. 2) и далее переходим к пространному обсуждению средств поддержки надежности нашей системы (разд. 3). Затем мы анализируем надежность (разд. 4) и эффективность (разд. 5) системы на основе реальных измерений. В конце статьи мы анализируем некоторые смежные работы (разд. 6) и представляем свои выводы (разд. 7).

2. Разработка операционной системы

Этот проект посвящен построению более надежной операционной системы. Прежде чем подробно описывать свою разработку, мы кратко обсудим, каким образом выбор структуры операционной системы может непосредственно влиять на ее надежность. В своих целях мы будем проводить различие между двумя структурами операционных систем: *монолитными системами* и *системами с минимальным ядром*. Существуют и другие типы операционных систем, такие как экзоядра [10] и виртуальные машины [24]. Они не имеют непосредственного отношения к данной статье, но мы вернемся к ним в разд 6.

2.1 Проблемы монолитных систем

Как показано на рис. 1, в стандартной монолитной системе ядро содержит все операционную систему, скомпонованную в едином адресном пространстве и выполняемую в режиме ядра. Ядро может быть структурировано на компоненты, или модули, показанные на рисунке в виде прямоугольников с пунктирными сторонами, но между компонентами отсутствуют защитные границы. В отличие от этого, прямоугольники со сплошными сторонами соответствуют отдельным процессам, выполняемым в режиме пользователя; каждый из этих процессов выполняется в отдельном адресном пространстве, защищаемом аппаратурой MMU (Memory Management Unit, устройство управления памятью).

С монолитными операционными системами связан ряд проблем, свойственных их архитектуре. Хотя некоторые из этих проблем уже упоминались во введении, мы приведем здесь их сводку:

1. Отсутствует должная изоляция сбоев.
2. Весь код выполняется на наивысшем уровне привилегированности.
3. Огромный размер кода предполагает наличие многочисленных ошибок.
4. В ядре присутствует ненадежный сторонний код.
5. Сложность систем затрудняет их сопровождение.

Этот список свойств ставит под сомнение надежность монолитных систем. Важно понимать, что эти свойства возникают не вследствие плохой реализации, а представляют собой фундаментальные проблемы, связанные с архитектурой операционной системы.

Предполагается корректность ядра, в то время, как только лишь его размер означает, что оно должно содержать многочисленные ошибки [27, 22, 2]. Более того, для всех операционных систем, в которых код выполняется на наивысшем уровне привилегированности, и не обеспечивается должное сдерживание распространения сбоев, любая ошибка может стать фатальной. Например, неправильно работающий драйвер устройства, предоставленный сторонним разработчиком, может легко разрушить ключевые структуры данных и вывести из строя всю систему. Реальность подобной угрозы следует из того наблюдения, что аварийные отказы большинства операционных систем случаются по вине драйверов устройств [7, 25]. Дополнительной проблемой является то, что огромный размер монолитных ядер делает их очень сложными и трудно понимаемыми. Без общего понимания ядра даже опытный программист может легко внести ошибки за счет недостаточной осведомленности о побочных эффектах своих

действий.

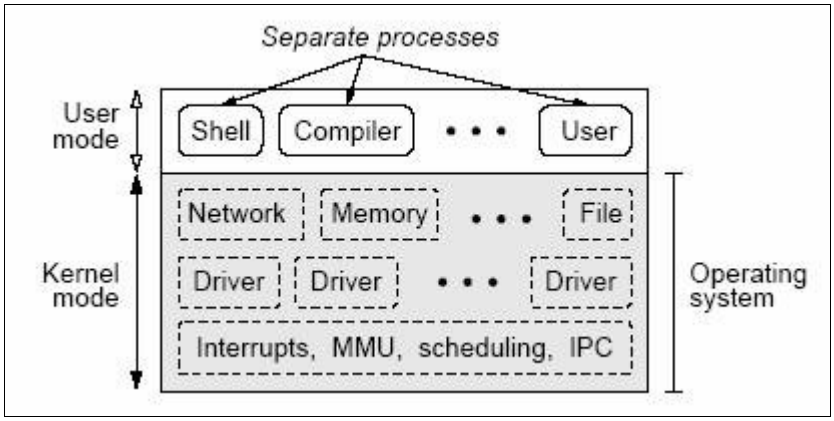

Рис. 1. Структура монолитной системы. Вся операционная система выполняется в режиме ядра без должной изоляции сбоев.

2.2 Системы с минимальным ядром

На другом полюсе находится минимальное ядро, содержащее только чистый механизм и никакой политики. Минимальное ядро включает обработчики прерываний, механизм для запуска и остановки процессов (путем загрузки регистров MMU и ЦП), планировщик и механизм поддержки межпроцессных коммуникаций; в идеальном случае больше в ядро не входит ничего. Поддержка функциональных возможностей стандартной операционной системы, представленных в монолитном ядре, перемещается в пользовательское адресное пространство, и соответствующий код больше не выполняется на наиболее привилегированном уровне.

Поверх минимального ядра возможны различные организации операционной системы. Одним из вариантов является выполнение всей операционной системы в одном сервере в пользовательском режиме, но в такой архитектуре существуют те же проблемы, что и в монолитной системе, и ошибки по-прежнему могут привести к аварийному отказу всей операционной системы, выполняемой в режиме пользователя. В разд. 6 мы обсудим некоторые работы в этой области.

Лучшим решением является выполнение каждого ненадежного модуля в пользовательском режиме в отдельном процессе, изолированном от других процессов. Мы до крайности увлеклись этой идеей и полностью раздробили свою систему, как показано на рис. 2. Все функциональные компоненты операционной системы, такие как драйверы устройств, файловая система, сервер сети и высокоуровневое управление памятью, выполняются как отдельные процессы в режиме пользователя в собственном адресном пространстве. Эту модель можно определить, как *мультисерверную операционную систему.*

С логической точки зрения наши пользовательские процессы можно разбить на три уровня, хотя с точки зрения ядра все они являются всего лишь процессами. Самый низкий уровень процессов, выполняемых в пользовательском режиме, занимают драйверы устройств, каждый из которых управляет некоторым устройством. Мы реализовали драйверы для интерфейса IDE, гибких и жестких дисков, клавиатуры, дисплеев, аудио-устройств, принтеров и различных карт Ethernet. Выше уровня драйверов находятся серверные процессы. В их число входят файловый сервер, сервер процессов, сетевой сервер, информационный сервер, сервер реинкарнации и другие. Над уровнем серверов выполняются обычные пользовательские процессы, включая различные интерпретаторы shell, компиляторы, утилиты и прикладные программы. Не считая небольшого числа исключений, серверы и драйверы являются нормальными пользовательскими процессами.

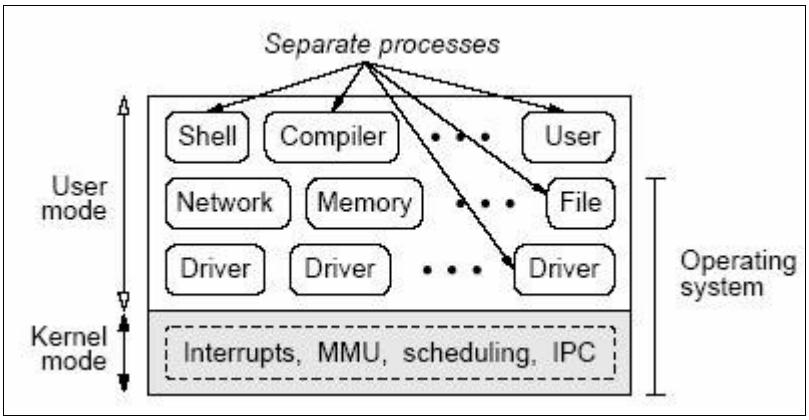

Рис. 2. Структура нашей системы. Операционная система выполняется в виде набора изолированных пользовательских процессов поверх крошечного ядра.

Во избежание какой-либо неясности еще раз заметим, что каждый сервер или драйвер выполняется в виде *отдельного* пользовательского процесса с собственным адресным пространством, полностью отделенным от адресного пространства ядра и других серверов, драйверов и процессов пользователей. В нашей архитектуре процессы не разделяют какое-либо адресное пространство и могут общаться друг с другом только с использованием механизма IPC, обеспечиваемого ядром. Этот аспект является критическим для надежности, поскольку он предотвращает распространение сбоев одного сервера или драйвера на другие серверы или драйверы подобно тому, как ошибка при компиляции программы, возникающая в одном процессе, не влияет на то, что делает браузер в другом процессе.

При работе в пользовательском режиме возможности процессов операционной системы ограничены. Поэтому для поддержки выполнения требуемых от них задач серверами и драйверами ядро экспортирует ряд системных вызовов, которые могут производиться авторизованными процессами. Например, драйверы устройств больше не имеют привилегий на непосредственное выполнение ввода-вывода, но могут запросить у ядра выполнения соответствующих действий от своего имени. Кроме того, серверы и драйверы могут запрашивать сервисы друг у друга. Все такие IPC производятся путем обмена небольшими сообщениями фиксированного размера. Этот обмен сообщениями реализуется путем обращений к ядру, которое до выполнения запрашиваемого действия проверяет, авторизован ли соответствующим образом вызывающий процесс.

Рассмотрим типичный вызов ядра. Компоненту операционной системы, выполняемому в пользовательском режиме в некотором процессе, может потребоваться скопировать данные в другое адресное пространство или из него, но ему невозможно доверить возможность доступа к физической памяти. Взамен этого обеспечиваются вызовы ядра для копирования из допустимых виртуальных адресов или в эти адреса сегмента данных целевого процесса. Этот вызов предоставляет гораздо более слабые возможности, чем запись в любое слово физической памяти, но все-таки эти возможности достаточно мощны, и поэтому возможность такого вызова предоставляется только процессам операционной системы, которым требуется копирование блоков данных из одного адресного пространства в другое. Для обычных пользовательских процессов подобные вызовы запрещены.

После приведения этого описания структуры операционной системы мы можем теперь объяснить, каким образом пользовательские процессы получают сервисы операционной системы, определенные в стандарте POSIX. Пользовательский процесс, желающий выполнить, например, вызов READ, формирует сообщение, содержащее номер системного вызова и (указатели на) параметры, и обращается к ядру с запросом посылки этого небольшого запросного сообщения файловому серверу, являющемуся другим пользовательским процессом. Ядро обеспечивает блокировку вызывающего процесса до тех пор, пока его запрос не будет обработан файловым сервером. По умолчанию все коммуникации между процессами запрещаются по соображениям безопасности, но этот запрос достигает цели, поскольку

коммуникации с файловым сервером явно разрешаются обычным пользовательским процессам.

Если запрашиваемые содержатся в буферном кэше файлового сервера, то он производит вызов ядра с запросом копирования этих данных в буфер пользователя. Если у файлового сервера отсутствуют требуемые данные, то он посылает сообщение дисковому драйверу с запросом нужного блока. Тогда дисковый драйвер выдает команду диску на чтение этого блока прямо по адресу внутри буферного кэша файлового сервера. Когда передача данных с диска завершается, дисковый драйвер посылает файловому серверу ответное сообщение, содержащее состояние запроса (успех или причина неудачи). После этого файловый сервер производит вызов ядра с запросом копирования блока в пользовательское адресное пространство.

Эта схема проста и элегантна; она позволяет отделить серверы и драйверы от ядра и позволяет заменять их простым образом, что способствует модульности системы. Хотя здесь требуется до четырех сообщений, они передаются очень быстро (в пределах 500 наносекунд на сообщение в зависимости от ЦП). Если и отправитель, и получатель готовы к коммуникации, то ядро копирует сообщение прямо из буфера отправитель в буфер получателя без его перемещения в адресное пространство ядра. Кроме того, число копирований данных является точно таким же, как в монолитной системе: диск помещает данные прямо в буферный кэш файлового сервера, и имеется одно копирование из этого кэша в адресное пространство пользовательского процесса.

2.3 Принципы разработки

Прежде чем перейти к подробному рассмотрению свойств надежности нашей системы, кратко обсудим принципы разработки, которыми мы руководствовались в стремлении к надежности:

1. Простота.
2. Модульность.
3. Наименьшая авторизация.
4. Отказоустойчивость.

Во-первых, мы сохраняем свою систему настолько простой, насколько это возможно, так что ее легко понять, и можно с большей вероятностью поддерживать ее в корректном состоянии. Это относится как к высокоуровневому проектированию, так и к реализации. Наша разработка позволяет структурно избежать известных проблем, таких как исчерпание ресурсов. При потребности мы явно обмениваем ресурсы и эффективность на надежность. Например, в ядре статически объявляются все структуры данных вместо того, чтобы динамически выделять память при необходимости. Хотя мы можем недоиспользовать некоторую память, этот подход является очень простым и никогда не приводит к ошибкам. Другим примером является то, что мы умышленно не реализовали нити. Может быть, мы заплатили за это некоторой потерей эффективности (а может быть, и нет), но зато не должны беспокоиться о потенциальных «состояниях гонок» (race condition) и синхронизации, что существенно облегчает жизнь программистам.

Во-вторых, мы разделили свою систему на набор небольших независимых модулей. Использование свойств модульности, таких как ограничение распространения сбоев, является ключевым элементом разработки нашей системы. Путем полного разделения операционной системы на модули мы можем установить «брандмаэры», сквозь которые не могут распространяться ошибки, что приводит к более надежной системе. Для предотвращения *косвенного* влияния сбоев в одном модуле на какой-либо другой модуль мы структурным образом уменьшаем их взаимозависимость, насколько это возможно. В тех случаях, когда это невозможно из-за природы модулей, мы применяем дополнительные средства поддержки безопасности. Например, файловая система зависит от драйверов устройств, но она

разрабатывается таким образом, чтобы быть готовой к обработке сбоев драйвера.

В третьих, мы обеспечиваем соблюдение принципа наименьшей авторизации. Хотя изоляция сбоев помогает сдерживать их распространение, сбой в полномочном модуле все еще может вызвать значительный ущерб. Поэтому мы понижаем уровень привилегий всех пользовательских процессов до предельно допустимого минимума. В ядре поддерживаются битовые массивы и списки, определяющие возможности процессов. В частности, имеются шкала допустимых вызовов ядра и список допустимых адресов назначения сообщений. Эта информация сохраняется в элементах таблицы процессов, и поэтому ее можно строго контролировать, и ею просто управлять. Информация об авторизации инициируется во время загрузки системы, главным образом, на основе конфигурационных таблиц, создаваемых системным администратором.

В четвертых, при разработке системы мы явным образом учитываем возможность к устойчивости к некоторым сбоям. Все серверы и драйверы управляются и отслеживаются специальным сервером, называемым *сервером реинкарнации*, который может справляться с двумя видами проблем. Если системный процесс завершается непредвиденным образом, это немедленно распознается, и процесс перезапускается. Кроме того, периодически проверяется состояние каждого системного процесса для проверки его правильного функционирования. Если процесс функционирует неправильно, он принудительно завершается и перезапускается. Так работает механизм отказоустойчивости: сбойный компонент заменяется, но система все время продолжает работать.

3. Свойства надежности

Мы считаем, что в нашей разработке надежность системы повышается по сравнению со всеми другими существующими операционными системами за счет применения трех важных подходов:

1. Уменьшается число критических сбоев.

2. Сокращается объем ущерба, который может быть причинен любой ошибкой.

3. Имеется возможность восстановления после распространенных сбоев.

В следующих подразделах мы объясним, почему применение этих подходов позволяет повысить надежность. Мы также сравним воздействие некоторых классов ошибок на нашу систему с тем, как они воздействуют на монолитные системы, такие как Windows, Linux и BSD. В разд. 6 мы сравним наш подход к повышению надежности с другими идеями, предлагаемыми в литературных источниках.

3.1 Сокращение числа ошибок в ядре

Нашей первой линией защиты является очень небольшое ядро. Хорошо известно, что в большем по объему коде содержится большее число ошибок, и поэтому чем меньше ядро, тем меньше в нем ошибок. Если в качестве нижней оценки использовать 6 ошибок на 1000 строк исполняемого кода [27], то при наличии 3800 строк исполняемого кода в ядре будет присутствовать, как минимум, 22 ошибки. Кроме того, 3800 строк кода (менее 100 страниц листинга, включая заголовки и комментарии) – это достаточно мало, чтобы весь этот код мог понять один человек; это существенно повышает шансы на то, что со временем все ошибки удастся найти.

В отличие от этого, в ядре монолитной системы, такой как Linux, размером в 2.5 миллиона строк исполняемого кода, вероятно, должно содержаться не менее 6 * 2500 = 15,000 ошибок. Кроме

того, при наличии системы из нескольких миллионов строк ни один человек не может прочитать весь исходный код и полностью понять, как он работает, что уменьшает шансы на нахождение всех ошибок.

3.2 Снижение потенциального влияния ошибок

Конечно, уменьшение размера ядра не приводит к сокращению объема всего кода системы. При этом всего лишь б*о*льшая часть системы начинает работать в пользовательском режиме. Однако само это изменение оказывает глубокое влияние на надежность. У кода ядра имеется возможность полного доступа ко всему, что может делать машина. Ошибки в ядре могут приводить к случайной инициализации ввода-вывода, выполнению неправильного ввода-вывода, повреждению таблиц распределения памяти и другим вещам, которые не могут сделать непривилегированные программы, выполняемые в пользовательском режиме.

Поэтому мы не утверждаем, что перевод большей части операционной системы в пользовательский режим приводит к сокращению общего числа имеющихся ошибок. Мы утверждаем лишь то, что эффект проявления ошибки при выполнении программы в пользовательском режиме является менее разрушительным, чем тот, который проявляется при выполнении программы в режиме ядра. Например, аудио-драйвер, выполняющийся в пользовательском режиме, при попытке использования неверного указателя насильственно завершается сервером процессов, аудио-аппаратура перестает работать, но на остальную часть системы это не оказывает влияния.

Для сравнения рассмотрим влияние ошибки в аудио-драйвере, выполняющемся в режиме ядра. Этот драйвер может непреднамеренно перезаписать в стеке адрес возврата из своей процедуры и совершить при выполнении возврата произвольный переход в монолитное ядро. Этот переход может привести в код управления памятью, вызывая разрушение ключевых структур данных, таких как таблицы страниц и списки свободных и занятых участков памяти. Монолитные системы в этом отношении являются очень хрупкими и легко разрушаются при проявлении ошибки.

3.3 Восстановление после сбоев

Серверы и драйверы запускаются и контролируются системным процессом, называемым *сервером реинкарнации*. Если контролируемый процесс непредвиденным или аварийным образом завершается, это немедленно распознается, поскольку сервер процессов оповещает сервер реинкарнации о завершении сервера или драйвера, и процесс автоматически перезапускается. Кроме того, сервер реинкарнации периодически опрашивает все серверы и драйверы на предмет их состояния. Если какой-либо из этих процессов не отвечает правильным образом в течение установленного интервала времени, то сервер реинкарнации насильственно завершает и перезапускает плохо ведущие себя серверы и драйверы. Поскольку очень многие ошибки ввода-вывода бывают неустойчивыми, проявляющимися при редко возникающих временных соотношениях, синхронизационных тупиках и т.д., простой перезапуск драйвера устраняет проблему.

Сбой драйвера имеет последствия и для файловой системы. Могут быть потеряны невыполненные запросы ввода-вывода, и в некоторых случаях информация об ошибке ввода-вывода доводится до сведения приложения. Однако во многих случаях возможно полное восстановление. Более подробное обсуждение сервера реинкарнации и надежности на уровне приложений приводится в разд. 4.

В монолитных системах обычно отсутствует возможность обнаружения сбойных драйверов «на лету», хотя имеются данные о некоторых исследованиях в этой области [25]. Тем не менее, замена на лету ядерного драйвера является сложным делом, поскольку ко времени замены он

может удерживать ядерные блокировки или находиться в критическом участке.

3.4 Ограничение злоупотреблений переполнениями буферов

Известно, что переполнения буферов являются обильным источником ошибок, наличием которых интенсивно пользуются вирусы и черви. Хотя наша разработка направлена скорее на борьбу с ошибками, а не со злоумышленным кодом, некоторые средства нашей системы предоставляют защиту от определенных видов злоупотреблений. Поскольку наше ядро является минимальным, и в нем используется только статическое размещение данных, возникновение проблемы маловероятно в наиболее чувствительной части системы. Если переполнение буфера случается в одном из пользовательских процессов, то проблема не является слишком серьезной, поскольку серверы и драйверы, выполняемые в пользовательском режиме, обладают ограниченными возможностями.

Кроме того, в нашей системе выполняется только код, расположенный в сегментах текста, которые доступны только по чтению. Хотя это не предотвращает возможность переполнений буферов, усложняется возможность злоупотребления, поскольку избыточные данные, находящиеся в стеке или куче, невозможно выполнить как код. Этот защитный механизм является исключительно важным, поскольку он предотвращает заражение вирусами и червями и выполнение их собственного кода. Сценарий наихудшего случая изменяется от взятия непосредственного управления до перезаписи адреса возврата в стеке и выполнения некоторой существующей библиотечной процедуры. Наиболее известный пример такой ситуации часто называют атакой путем «возврата в libc» («return-to-libc»), и этот способ атаки считается гораздо более сложным, чем выполнение кода в стеке или куче.

В отличие от этого, в монолитных системах приобретаются привилегии суперпользователя, если переполнение буфера происходит в любой части операционной системы. Более того, во многих монолитных системах допускается выполнение кода в стеке или куче, что существенно упрощает злоупотребление переполнениями буферов.

3.5 Обеспечение надежного IPC

Хорошо известной проблемой механизмов обмена сообщениями является управление буферами, но в нашем варианте коммуникационных примитивов мы полностью избегаем этой проблемы. В нашем механизме синхронной передачи сообщений используются рандеву, в результате чего устраняется потребность в буферизации и управлении буферами, а также отсутствует проблема исчерпания ресурсов. Если получатель не ожидает сообщения, то примитив SEND блокирует отправителя. Аналогично, примитив RECEIVE блокирует процесс, если отсутствует сообщение, ожидающее своего получения. Это означает, что для заданного процесса в таблице процессов в любое время должен храниться единственный указатель на буфер сообщения.

В дополнение к этому, у нас имеется механизм асинхронной передачи сообщений NOTIFY, который также не является чувствительным к исчерпанию ресурсов. Оповестительные сообщения являются типизированными, и для каждого процесса сохраняется только один бит для каждого типа. Хотя объем информации, которую можно передать таким образом, ограничен, этот подход был выбран из-за своей надежности.

Кстати, заметим, что в своем IPC мы избегаем переполнений буферов путем ограничения средств коммуникации короткими сообщениями фиксированной длины. Сообщение является объединением нескольких типизированных форматов сообщений, так что размер автоматически выбирается компилятором, как размер наибольшего допустимого типа сообщений, который зависит от размера целых чисел и указателей. Этот механизм передачи сообщений используется для всех запросов и ответов.

3.6 Ограничение IPC

IPC – это мощный механизмом, который нуждается в строгом контроле. Поскольку наш механизм передачи сообщений является синхронным, процесс, выполняющий примитив IPC, блокируется, пока оба участника не станут готовыми. Пользовательский процесс может легко злоупотребить этим свойством для завешивания системных процессов путем посылки запроса без ожидания ответа. Поэтому имеется другой примитив IPC SENDREC, комбинирующий в одном вызове SEND и RECEIVE. Он блокирует отправителя до получения ответа на запрос. С целью защиты операционной системы этот примитив является единственным, который можно использовать обычным пользователям. В действительности, в ядре для каждого процесса поддерживается битовый массив для ограничения примитивов IPC, которые позволяется использовать данному процессу.

Кроме того, в ядре поддерживается битовый массив, определяющий, с какими драйверами и серверами может взаимодействовать данный процесс. Эта маска посылки сообщений представляет собой механизм, предотвращающий непосредственную посылку сообщений драйверам от пользовательских процессов. Взамен этого, им разрешается общаться только с серверами, обеспечивающими POSIX-вызовы. Однако маска посылки сообщений используется также и для предотвращения посылки (непредусмотренного) сообщения, скажем, от драйвера клавиатуры аудио-драйверу. Снова путем строгой инкапсуляции возможностей каждого процесса мы можем в значительной степени предотвратить распространение неминуемых ошибок в драйверах и их воздействие на другие части системы.

В отличие от этого, в монолитной системе любой драйвер может вызвать любой кусок кода в ядре, используя машинную инструкцию вызова подпрограммы (или, еще хуже, инструкцию возврата из подпрограммы, если стек был перезаписан по причине переполнения буфера), что позволяет проблемам, возникающим в одной подсистеме, распространяться в другие подсистемы.

3.7 Избегание тупиков

Поскольку по умолчанию для IPC используются синхронные вызовы SEND и RECEIVE, могут возникать тупики, когда два или большее число процессов одновременно пытаются обмениваться сообщениями, и все процессы блокируются в ожидании один другого. Поэтому мы тщательно разрабатывали протокол *избегания тупиков*, предписывающий частичное, нисходящее упорядочение сообщений.

Упорядочение сообщений приблизительно соответствует разбиению на уровни, описанному в разд. 2.2. Например, обычным пользовательским процессам позволяется только посылать сообщения с использованием примитива SENDREC серверам, реализующим интерфейс POSIX, а эти серверы могут запрашивать сервисы от драйверов, которые, в свою очередь, могут производить вызовы ядра. Однако для асинхронных событий, таких как прерывания и таймеры, требуются сообщения, посылаемые в противоположном направлении, от ядра серверу или драйверу. Использование синхронных вызовов SEND для передачи этих событий может легко привести к тупику. Мы избегаем этой проблемы путем использования для асинхронных событий механизма NOTIFY, который никогда не блокирует вызывающую сторону. Если оповестительное сообщение не может быть доставлено процессу-адресату, оно сохраняется в его элементе таблицы процессов до тех пор, пока он не выполнит RECEIVE.

Хотя протокол избегания тупиков поддерживается обсуждавшимся выше механизмом масок посылки сообщений, мы также реализовали в ядре *распознавание тупиков.* Если вызов примитива в некотором процессе непредусмотренным образом привел бы к возникновению тупика, то выполнение примитива не производится, и вызывающей стороне возвращается

сообщение об ошибке.

3.8 Унификация прерываний и сообщений

Базовым механизмом IPC является передача сообщений на основе рандеву, но требуются и асинхронные сообщения, например, для предоставления информации о прерываниях, что является потенциальным источником ошибок в операционных системах. Мы существенно уменьшили здесь шансы на появление ошибок, унифицировав асинхронные сигналы и сообщения. Обычно, когда некоторый процесс посылает сообщение другому процессу и получатель не является готовым, отправитель блокируется. Эта схема не работает для прерываний, поскольку обработчик прерываний не может позволить себе блокировку. Вместо этого используется асинхронный механизм оповещений, при использовании которого обработчик прерываний производит вызов NOTIFY для драйвера. Если драйвер ожидает сообщение, то оповещение доставляется напрямую. Если он его не ожидает, то оповещение сохраняется в битовом массиве до тех пор, пока впоследствии драйвер не выполнит вызов RECEIVE.

3.9 Ограничение функциональных возможностей драйвера

Ядро экспортирует ограниченный набор функций, которые можно вызывать извне. Этот ядерный API представляет собой единственный способ взаимодействия драйвера с ядром. Однако не каждому драйверу разрешается использовать любой вызов ядра. Для каждого драйвера в ядре (в таблице процессов) поддерживается битовый массив, показывающий, какие вызовы ядра может производить этот драйвер. Гранулярность вызовов ядра является достаточно мелкой. Отсутствует мультиплексирование вызовов в один и тот же номер функции. Каждый вызов индивидуально защищается собственным битом в битовом массиве. Однако на внутреннем уровне несколько вызовов может обрабатываться одной и той же ядерной функцией. Этот метод позволяет реализовать детальное управление доступом к ядру.

Например, некоторым драйверам требуется доступ по чтению и записи к данным, находящимся в пользовательских адресных пространствах, но вызовы для чтения и записи в этих пространствах являются разными. Так что мы не мультиплексируем чтение и запись в один вызов с использованием параметра «направление». Соответственно, можно разрешить, например, драйверу принтера выполнять вызов ядра для чтения данных из пользовательских процессов, но не разрешать выполнение вызовов для записи. Вследствие этого ошибка в драйвере, которому разрешено только чтение, не может привести к случайному повреждению пользовательского адресного пространства.

Сравним эту ситуацию с возможным поведением драйвера устройства в монолитном ядре. Ошибка в коде может привести к записи в адресное пространство пользовательского процесса вместо чтения из него, что разрушит процесс. Кроме того, ядерный драйвер может вызвать любую функцию во всем ядре, включая функции, которые не должны вызываться драйверами. Поскольку отсутствует какая-либо внутриядерная защита, это практически невозможно предотвратить. В нашей разработке ни один драйвер не может вызвать ядерную функцию, которая не была явно экспортирована как часть интерфейса между ядром и этим драйвером.

3.10 Запрещение доступа к портам ввода-вывода

Для каждого драйвера в ядре поддерживается список портов ввода-вывода, из которых он может читать, а также тех, в которые он может писать. Чтение и запись защищаются по отдельности, так что процесс, у которого имеется право на только чтение из некоторого порта ввода-вывода, не может писать в него. Любая попытка нарушения этих правил приводит к выработке кода ошибки, возвращаемого вызывающей стороне. Таким образом, драйвер принтера может быть

ограничен доступом только к портам ввода-вывода принтера, аудио-драйвер может быть ограничен доступом только к портам ввода-вывода звуковой карты и т.д.

В отличие от этого, в монолитных системах отсутствует способ ограничения доступа внутриядерного драйвера только к небольшому числу портов ввода-вывода. Ядерный драйвер может случайно произвести запись в любой порт ввода-вывода и нанести существенный ущерб.

В некоторых случаях в адресное пространство драйвера могут отображаться реальные регистры устройства ввода-вывода, чтобы избежать какого бы то ни было взаимодействия с ядром при совершении ввода-вывода. Однако, поскольку не во всех архитектурах допускается отображение регистров ввода-вывода в пользовательские процессы с обеспечением требуемого уровня защиты, мы выбрали модель, в которой реальные операции ввода-вывода выполняются только ядром. Это проектное решение является еще одним примером того, что мы отдаем предпочтение надежности в ущерб эффективности.

Хотя в настоящее время таблицы, разрешающие доступ к портам ввода-вывода, инициализируются из конфигурационного файла, мы планируем реализовать сервер шины PCI, который будет делать это автоматически. Сервер шины PCI может получить из BIOS порты ввода-вывода, требуемые каждому драйверу, и использовать эту информацию для инициализации таблиц ядра.

3.11 Проверка параметров

Поскольку все вызовы ядра производятся путем генерации внутреннего прерывания, ядро может выполнить ограниченную валидацию параметров до диспетчеризации вызова. Эта валидация включает проверки как *исправности (sanity)*, так и *прав доступа* (*permission)*. Например, если драйвер просит ядро записать блок данных с использованием физической адресации, то этот вызов может быть отвергнут, поскольку не у всех драйверов имеется право на такие действия. Используя виртуальную адресацию, ядро, по-видимому, не сможет сказать, является ли этот адрес записи правильным, но оно, по крайней мере, сможет проверить, что этот адрес действительно является допустимым адресом в сегменте данных или стека пользовательского процесса, а не относится к сегменту текста и не является каким-то случайным недействительным адресом.

Хотя такие проверки исправности являются грубыми, это лучше, чем ничего. В монолитных системах ничто не препятствует драйверу выполнять запись по адресам, по которым нельзя писать не при каких условиях, таким как адреса в сегменте текста ядра.

3.12 Отлавливание плохих указателей

В программах на языках C и C++ используется множество указателей, и эти программы все время подвержены ошибкам, связанным с использованием плохих указателей. Разыменование неверного указателя часто приводит к выявлению аппаратурой ошибки сегментации. В нашей разработке сервер или драйвер, пытающиеся разыменовать плохой указатель, принудительно завершаются, и выдается дамп памяти для будущей отладки, точно так же, как и для других пользовательских процессов. Если плохой указатель обнаруживается в части операционной системы, выполняемой в пользовательском режиме, то сервер реинкарнации немедленно замечает наличие сбойной ситуации и заменяет принудительно завершенный процесс его свежей копией.

3.13 Укрощение бесконечных циклов

Когда драйвер впадает в бесконечный цикл, это создает угрозу потребления бесконечного

времени ЦП. Планировщик замечает наличие такого поведения и постепенно понижает приоритет неисправного процесса, пока он не становится неработающим процессом. Однако другие процессы могут продолжать нормально работать. После исчерпания предопределенного интервала времени сервер реинкарняции заметит, что данный драйвер не отвечает на запросы, принудительно завершит и перезапустит его.

В отличие от этого, когда в бесконечный цикл впадает ядерный драйвер, он потребляет все время ЦП и фактически завешивает всю систему.

3.14 Проверка DMA

Одной из вещей, которую мы не можем обеспечить, является предотвращение причинения ущерба системе по причине неверного DMA (Direct Memory Access, прямой доступ к памяти). Для предотвращения перезаписи драйвером через DMA произвольной части реальной памяти требуется аппаратная защита. Однако мы можем обнаружить некоторые ошибки DMA следующим образом. DMA обычно запускается путем записи адреса DMA в некоторый порт ввода-вывода. Мы можем предоставить библиотечную процедуру, которая вызывается для записи в некоторый порт ввода-вывода с предварительным декодированием (способом, зависящим от устройства) записей в этот порт ввода-вывода с целью нахождения используемых адресов DMA и проверки их допустимости. В злонамеренных драйверах такая проверка может обходиться, но в добропорядочных драйверах этот способ позволяет выловить хотя бы некоторые ошибки при умеренных накладных расходах.

В зависимости от аппаратуры мы можем поступить еще лучше. Если бы на периферийной шине имелось MMU (Memory Management Unit, устройство управления памятью) ввода-вывода, мы могли бы точно ограничить доступ к памяти для каждого драйвера [16]. Для систем с шиной PCI-X мы собираемся возложить на свой сервер шины PCI ответственность за инициализацию таблиц MMU ввода-вывода. Это часть нашей будущей работы.

4. Анализ надежности

Для проверки надежности системы мы вручную внесли некоторые тщательно подобранные ошибки в некоторые из своих серверов и драйверов, чтобы посмотреть, что в результате произойдет. Как описывалось в разд. 3.3, наша система разрабатывается для обнаружения и исправления многих ошибок, и именно это мы и наблюдали. Если по какой бы то ни было причине происходил сбой некоторого компонента, это распознавалось сервером реинкарнации, который применял все требуемые средства для оживления сбойного компонента. Ниже это описывается более подробно.

Для понимания работы нашей системы нужно различать два класса ошибок. Первый класс составляют *логические ошибки*, означающие, что сервер или драйвер придерживается протокола межмодульных взаимодействий и нормально отвечает на запросы, как если бы он успешно выполнил работу, чего в действительности не происходит. Примером является драйвер принтера, который печатает бессмысленную информацию, но производит нормальные возвраты. Для *любой* системы очень трудно, если не невозможно, отлавливать ошибки такого рода. Логические ошибки находятся за пределами этого исследования.

Второй класс состоит из *протокольных ошибок*, при наличии которых нарушаются правила, определяющие поведение серверов и драйверов. Например, в нашей системе от серверов и драйверов требуется отвечать на периодические запросы состояния, поступающие от сервера реинкарнации. Если они не подчиняются этому правилу, предпринимается корректирующее действие. Наша система разрабатывается для борьбы с протокольными ошибками.

4.1 Сервер реинкарнации

Сервер реинкарнации – это центральный сервер, управляющий всеми серверами и драйверами операционной системы. Он позволяет существенно повысить надежность, обеспечивая:

1. Немедленное распознавание фатальных сбоев.

2. Периодический мониторинг состояния.

Таким образом, он помогает отлавливать два распространенных вида сбоев: умершие или плохо себя ведущие системные процессы и незамедлительно принимается за решение наиболее острой проблемы. Если системный процесс завершается, то сервер реинкарнации напрямую оповещается об этом и проверяет свои таблицы, чтобы понять, следует ли перезапустить сервис. Этот механизм, например, обеспечивает незамедлительную замену драйвера, принудительно завершенного по причине использования плохого указателя. Кроме того, периодический мониторинг состояния помогает дисциплинировать плохо себя ведущие системные сервисы. Например, драйвер, который впадает в бесконечный цикл и не может ответить на запрос состояния от сервера реинкарнации, будет принудительно завершен и перезапущен.

Замена драйвера устройства состоит из строго контролируемой последовательности действий. Во-первых, сервер реинкарнации порождает новый процесс, выполнение которого задерживается, поскольку для него еще не назначены привилегии. Затем сервер реинкарнации сообщает о новом драйвере файловой системе и, наконец, назначает требуемые привилегии. Когда все эти шаги успешно выполняются, новый процесс начинает работать и выполняет код драйвера, берущийся из файловой системы. В качестве дополнительной предосторожности двоичный код некоторых драйверов может дублироваться в основной памяти, чтобы, например, драйвер для диска корневой файловой системы можно было загрузить без потребности в обмене с диском.

4.2 Надежность уровня приложений

Наличие сбойного драйвера может приводить к последствиям для файловой системы и приложений, производящих ввод-вывод. Если у файловой системы имелся невыполненный запрос ввода-вывода, ей будет возвращен код ошибки, говорящий о сбое драйвера. В этот момент могут быть предприняты различные действия. Необходимо проводить различие между блочными и символьными устройствами, потому что ввод-вывод для блочных устройств буферизуется в буферном кэше файловой системы. На рис. 3 приводится обзор различных сценариев восстановления на уровне приложения.

Driver	Type	Recovery	How
Hard disk	Block	Full	Flush FS cache
RAM disk	Block	Full	Flush FS cache
Floppy	Block	Full	Flush FS cache
Printer	Character	Partial	Reissue print job
Ethernet	Character	Full	Transport layer
Sound	Character	Partial	Jitter

Рис. 3. Различные сценарии восстановления на уровне приложений для разных типов сбойных драйверов устройств

При фатальном сбое блочного драйвера возможно полное восстановление без потери данных, прозрачное для приложения. Когда распознается сбой, сервер реинкарнации запускает новую копию драйвера и сбрасывает кэш файловой системы для синхронизации. Таким образом,

буферный кэш не только повышает производительность, но также является важным и для надежности.

Прозрачное восстановление иногда является возможным и при сбоях драйверов символьных устройств. Поскольку запрос ввода-вывода не буферизуется в кэше блоков файловой системы, информация об ошибке ввода-вывода должна быть доведена до приложения. Если приложение не может произвести восстановление, о проблеме будет оповещен пользователь. Фактически, сбои драйверов проталкиваются наверх, что приводит к различным сценариям восстановления. Например, если происходит сбой драйвера Ethernet, то сетевой сервер заметит отсутствие пакетов и произведет прозрачное восстановление, если приложение использует надежный транспортный протокол, такой как TCP. С другой стороны, если происходит сбой драйвера принтера, то пользователь, конечно, заметит, что его вывод на печать не удался и повторит команду печати.

Таким образом, во многих случаях наша система может обеспечить *полное* восстановление на прикладном уровне. В оставшихся случаях информация о сбоях ввода-вывода доводится до пользователя. Можно было бы смягчить это неудобство путем использования теневого драйвера для восстановления приложений, который использовали сбойный драйвер в момент его фатального сбоя, применяя методы, продемонстрированные в [25]. Нам не дает сделать это недостаток рабочей силы.

4.3 Результаты проверки надежности

Для проверки надежности своей системы мы вручную внесли сбои в некоторые из своих драйверов, чтобы протестировать некоторые виды ошибок и посмотреть на то, что получится. В простейшем случае мы завершали драйвер с применением сигнала SIGKILL. Более серьезные тестовые случаи вынуждали драйверы разыменовывать плохие указатели или впадать в бесконечный цикл. Во всех случаях сервер реинкарнации распознавал проблему и заменял неисправный драйвер свежей копией. Результаты тестов показаны на рис. 4.

Cause	Effect	Action	Recovery
Bad pointer	Killed	Restart	OK
Infinite loop	Hung	Kill & Restart	OK
Panic	Exit	Restart	OK
Kill signal	Killed	Restart	OK

Рис. 4. Результаты тестирования серьезных сбоев в драйверах устройств. Если сервер реинкарнации распознает проблему, он автоматически предпринимает корректирующие действия

Из тестировании надежности мы извлекли несколько уроков, важных для разработки нашей системы. Во-первых, поскольку сервер реинкарнации перезапускает неисправные серверы и драйверы, требуется, чтобы в них не сохранялось состояние, и они могли бы быть должным образом повторно инициализированы при повторном запуске. Компоненты, сохраняющие состояние, такие как файловая система и сервер процессов, невозможно излечить таким образом, поскольку они слишком много теряют при перезапуске. Наши возможности ограничены.

Другое наблюдение состоит в том, что некоторые драйверы были реализованы таким образом, что инициализация происходит только при первом вызове OPEN. Однако для прозрачного восстановления после сбоя драйвера на уровне приложений не должен требоваться повторный вызов OPEN. Вместо этого, выполнение вызова READ или WRITE в восстановленном драйвере должно заставить драйвер произвести повторную инициализацию.

Кроме того, хотя мы признаем наличие зависимостей между файловой системой и драйверами, наши тесты выявили некоторые другие взаимозависимости. Например, наш информационный сервер, выдающий на экран отладочные дампы при нажатии функциональных клавиш, теряет

свое отображение клавиш после перезапуска. В качестве общего правила, зависимости следует предотвращать, и все компоненты должны быть подготовлены для борьбы с непредусмотренными сбоями.

Наконец, чтобы еще больше повысить надежность, следует изменить и пользовательские приложения. По историческим причинам в большинстве приложений предполагается, что любой сбой драйвера является фатальным, и они немедленно сдаются, хотя иногда возможно восстановление. Примером, в котором возможно восстановление на уровне приложения, является печать. Если демон линейного принтера извещается о временном сбое драйвера, он может автоматически повторно выдать команду печати без вмешательства пользователя. Дальнейшие эксперименты с восстановлением на уровне приложений являются частью нашей будущей работы.

5. Измерения производительности

Производительность является проблемой, сопутствующей минимальным ядрам на протяжении десятилетий. Поэтому немедленно встает вопрос: во что обходятся обсуждавшиеся выше изменения? Чтобы разобраться в этом, мы создали прототип, состоящий из небольшого ядра и поддерживаемого им набора драйверов устройств и серверов, работающих в режиме пользователя. В качестве основы прототипа мы начали с использования системы MINIX 2 из-за ее небольшого размера и долгой истории. Код системы изучался многими десятками тысяч студентов в сотнях университетов в течение 18 лет, и в последние 10 лет почти не поступали сообщения об ошибках, имеющих отношение к ядру; по-видимому, отсутствие ошибок связано с малыми размерами ядра. Затем мы значительно изменили код, удалив из ядра драйверы устройств и добавив средства повышения надежности, обсуждавшиеся в разд. 3. Таким образом, мы получили практически новую систему MINIX 3 без потребности в написании большого объема кода, не существенного для данного проекта, такого как драйверы и файловая система.

Поскольку нас интересует стоимость изменений, обсуждавшихся в данной статье, мы сравниваем свою систему с базовой системой, в которой драйверы устройств являются частью ядра, путем запуска одних и тех же тестов на обеих системах. Это гораздо более чистая проверка, чем сравнение нашей системы с Linux или Windows, которое напоминало бы сравнение яблок с ананасами. Таким сравнениям часто мешают различия в качестве компиляторов, в стратегиях управления памятью, в файловых системах, в объеме выполненной оптимизации, в зрелости систем и во многих других факторах, которые могут полностью затенить все остальное.

Тестовой системой был 2.2 GHz Athlon (более точно, AMD64 3200+) с 1 Гб основной памяти и 40 гигабайтным диском IDE. Ни один из драйверов не был оптимизирован для работы в пользовательском режиме. Например, мы ожидаем, что на Pentium сможем обеспечить защищенным образом прямой доступ драйверов устройств к требуемым им портам ввода-вывода, устраняя, таким образом, многие вызовы ядра. Однако для поддержания переносимости интерфейс не будет изменяться. Кроме того, в настоящее время в драйверах используется программируемый ввод-вывод, что гораздо медленнее использования DMA. После реализации этих оптимизаций мы ожидаем существенного повышения эффективности. Тем не менее, даже при использовании существующей системы ухудшение производительности оказалось вполне разумным.

5.1 Результаты тестирования системных вызовов

Первый пакет тестов содержал тесты чистых POSIX-совместимых системных вызовов. Пользовательская программа должна была зафиксировать реальное время в тактах системных часов (на частоте 60 Гц), затем миллионы раз произвести системный вызов, после чего снова зафиксировать реальное время. Время обработки системного вызова вычислялось как разность

между конечным и начальным временем, деленная на число вызовов, за вычетом накладных расходов на организацию цикла, которые измерялись отдельно. Число итераций цикла было разным для каждого теста, поскольку тестирование 100 миллионов раз вызова getpid было разумным, но чтение 100 миллионов раз из 64-магабайтного файла заняло бы слишком много времени. Все тесты выполнялись на незагруженной системе. Для этих тестов частоты успешных обращений к кэшу ЦП и кэшу файлового сервера предположительно составляли 100%. Результаты показаны на рис. 5.

Call	Kernel	User	Δ	Ratio
getpid	0.831	1.011	0.180	1.22
lseek	0.721	0.797	0.076	1.11
open+close	3.048	3.315	0.267	1.09
read 64k+lseek	81.207	87.999	6.792	1.08
write 64k+lseek	80.165	86.832	6.667	1.08
creat+wr+del	12.465	13.465	1.000	1.08
fork	10.499	12.399	1.900	1.18
fork+exec	38.832	43.365	4.533	1.12
mkdir+rmdir	13.357	14.265	0.908	1.07
rename	5.852	6.812	0.960	1.16
Average				**1.12**

Рис. 5. Время системных вызовов для драйверов, выполняемых в режиме ядра, и драйверов, выполняемых в пользовательском режиме. Все значения времени представлены в микросекундах.

Кратко проанализируем результаты этих тестов. Выполнение системного вызова getpid заняло 0.831 мсек при использовании ядерных драйверов и 1.011 мсек при использовании драйверов, работающих в режиме пользователя. При выполнении этого вызова от пользовательского процесса менеджеру памяти посылается одиночное сообщение, на которое немедленно получается ответ. При использовании драйверов, выполняемых в режиме пользователя, вызов выполняется медленнее из-за наличия проверки прав процессов на посылку таких сообщений. При выполнении такого простого вызова существенное замедление вызывают даже несколько дополнительных строк кода. Хотя в процентах разница составляет 22%, на каждый вызов тратится лишь 180 дополнительных наносекунд, так что даже при частоте 10,000 обращений в секунду потери составляют всего 2.2 мсек в секунду, гораздо меньше 1%. При выполнении вызова lseek производится гораздо большая работа, и поэтому относительные накладные расходы снижаются до 11%. При выполнении открытия и закрытия файла этот показатель составляет всего 9%.

Чтение и запись 64-килобайтных участков данных занимает менее 90 мсек, и падение производительности составляет 8%. При использовании драйверов, выполняющихся в режиме пользователя, создание файла, запись в него 1 килобайта данных и удаление этих данных занимают 13.465 мсек. Из-за использования буферного кэша файлового сервера ни в одном из этих тестов не вызывались драйверы, и поэтому мы можем заключить, что другие изменения, не связанные с драйверами, замедляют систему примерно на 12%.

5.2 Результаты тестирования дискового ввода-вывода

Во втором пакете тестов мы читали из файла и писали в файл порции от 1 килобайта до 64 мегабайт. Тесты пропускались много раз, так что читаемый файл размещался в 12-мегабайтном кэше файлового сервера, кроме случая 64-мегабайтных обменов, когда объема кэша не хватало. Использование внутреннего кэша дискового контроллера не блокировалось. Результаты

показаны на рис. 6.

File reads	2.0.4	3.0.1	Δ	Ratio
1 KB	2.619	2.904	0.285	1.11
16 KB	18.891	20.728	1.837	1.10
256 KB	325.507	351.636	26.129	1.08
4 MB	6962.240	7363.498	401.258	1.06
64 MB	16.907	17.749	0.841	1.05
Average				**1.08**

File writes	2.0.4	3.0.1	Δ	Ratio
1 KB	2.547	3.004	0.457	1.18
16 KB	18.593	20.609	2.016	1.11
256 KB	320.960	345.696	24.736	1.08
4 MB	8376.329	8747.723	371.394	1.04
64 MB	18.789	19.294	0.505	1.03
Average				**1.09**

Рис. 6. Время чтения и записи порций большого файла. Значения времени приводятся в микросекундах, кроме 64-мегабайтных операций, для которых время указывается в секундах.

Как мы видим, разница в производительности составляет от 3% до 18%, в среднем – 8.4%. Однако заметим, что худший показатель производительности получен для 1-килобайтных записей, но абсолютное время возросло всего на 457 наносекунд. Это соотношение уменьшается при увеличении объема ввода-вывода, поскольку сокращаются относительные накладные расходы. В трех 64-магабайтных тестах, результаты которых показаны на рис. 6 и 7, это соотношение составляет всего от 3% до 5%.

В другом тесте производится чтение из непосредственного блочного устройства, соответствующего жесткому диску. Запись на непосредственное устройство разрушила бы его содержимое, поэтому такой тест не выполнялся. Результаты показаны на рис. 7. При выполнении этих тестов не используется буферный кэш файловой системы, и проверяется только перемещение битов с диска. Как мы видим, в этом случае средний показатель накладных расходов составляет всего 9%.

Raw reads	2.0.4	3.0.1	Δ	Ratio
1 KB	2.602	2.965	0.363	1.14
16 KB	17.907	19.968	2.061	1.12
256 KB	303.749	332.246	28.497	1.09
4 MB	6184.568	6625.107	440.539	1.07
64 MB	16.729	17.599	0.870	1.05
Average				**1.09**

Рис. 7. Время чтения из непосредственного дискового блочного устройства. Значения времени приводятся в микросекундах, кроме 64-мегабайтных операций, для которых время указывается в секундах.

5.3 Результаты тестирования приложений

Следующий набор тестов состоял из реальных программ, а не простых измерений времени выполнения системных вызовов. Результаты приведены на рис. 8. Первый тест состоял в

построении области начальной загрузки (boot image) в цикле, содержащем вызов system(”make image”); тем самым, построение производилось много раз. При каждом построении компилятор языка C вызывался 123 раза, ассемблер – 4 раза и компоновщик – 11 раз. Построение ядра, драйверов, серверов и программы *init*, а также сборка области начальной загрузки заняли 3.878 секунд. Среднее время компиляции составляло 32 мсек на файл.

Program	2.0.4	3.0.1	Δ	Ratio
Build image	3.630	3.878	0.248	1.07
Build POSIX tests	1.455	1.577	0.122	1.08
Sort	99.2	103.4	4.2	1.04
Sed	17.7	18.8	1.1	1.06
Grep	13.7	13.9	0.2	1.01
Prep	145.6	159.3	13.7	1.09
Uuencode	19.6	21.2	1.6	1.08
Average				**1.06**

Рис. 8. Время выполнения в секундах различных тестовых программ. Первые два теста выполнялись в цикле, а остальные пропускались только по одному разу для исключения влияния со стороны кэша файловой системы.

Второй тест содержал цикл, в котором компилировались тесты соответствия стандарту POSIX. Набор из 42 тестовых программ компилировался за 1,577 секунды, или примерно за 37 мсек на файл теста. Тесты с третьего по седьмой состояли в сортировке к 64-мегабайтного файла и применении к нему sed, grep, prep и uuencode соответственно. В этих тестах в разных объемах смешивались вычисления и обмены с диском. Каждый тест пропускался только по одному разу, так что кэш файловой системы практически не использовался; каждый блок брался с диска. Среднее падение производительности составило в этих случаях 6%, аналогично последним строчкам на рис. 6 и 7.

Если взять среднее значение для последнего столбца показателей 22 тестов, отраженных на рис. 6-8, мы получим 1.08. Другими словами, версия с драйверами, выполняемыми в режиме пользователя, оказалась примерно на 8% медленнее версии с ядерными драйверами для операций, вовлекающих обмены с дисками.

5.4 Сетевая производительность

Мы тестировали также и сетевую производительность системы с драйверами, выполняемыми в режиме пользователя. Тестирование производилось с использованием карты Intel Pro/100, поскольку у нас не было драйвера для карты Intel Pro/1000. Мы смогли управлять Ethernet на полной скорости. Кроме того, мы запускали тесты возвратной петли с отправителем и получателем, находящимися на одной машине, и наблюдали пропускную способность в 1.7 Гб/сек. Поскольку это эквивалентно использованию сетевого соединения для посылки на скорости 1.7 Гб/сек и одновременного приема на той же скорости, мы уверены, что управление гигабитной аппаратурой Ethernet с единственным однонаправленным потоком на скорости в 1 Гб/сек не должно создать проблему при использовании драйвера, выполняемого в режиме пользователя.

5.5 Размер кода

Скорость – это не единственный показатель, представляющий интерес; очень важным является и число ошибок. К сожалению, мы не можем напрямую пересчитать все ошибки, но разумным заменителем числа ошибок, вероятно, является число строк кода. Напомним: чем больше код, тем больше ошибок.

Подсчитать число строк кода не так просто, как может показаться на первый взгляд. Во-первых, пустые строки и комментарии не добавляют в код сложности, и поэтому мы их не учитываем. Во-вторых, #define и другие определения в файлах заголовков также не добавляют в код сложности, и поэтому файлы заголовков тоже не учитываются. Подсчет числа строк выполнялся с использованием Perl-скрипта *sclc.pl*, доступного в Internet. Результаты для ядра, четырех серверов (файловой системы, сервера процессов, сервера реинкарнации, информационного сервера), пяти драйверов (жесткого диска, флоппи-диска, RAM-диска, терминала, устройства журнализации) и программы *init* показаны на рис. 9.

На рисунке можно видеть, что ядро состоит из 2947 строк на языке С и 778 строк на языке ассемблера (для программирования низкоуровневых функциональных возможностей, таких как перехват прерываний и сохранение регистров ЦП при переключении процессов). Всего имеется 3725 строк кода. И *только* этот код исполняется в режиме ядра. Другим способом измерения размера кода для С-программ является подсчет числа точек с запятой, поскольку многие операторы языка С завершаются точкой с запятой. В коде ядра имеется 1729 точек с запятой. Наконец, размер откомпилированного ядра составляет 21,312 байт. Это число задает только размер кода (т.е. сегмента текста). Инициализированные данные (3800 байт) и стек в это число не входят.

Part	#	C	Asm	;	Binary
Init	1	327	0	193	7088
File	25	4648	0	2698	43,056
Process	13	2242	0	1308	20,208
Reinc.	28	519	0	278	6368
Info	6	783	0	457	13,808
Hard disk	1	1192	0	653	24,384
Floppy	1	770	0	435	10,448
RAM disk	1	237	0	116	4992
Terminal	19	5161	120	2120	26,352
Log device	4	430	0	235	6048
Kernel	45	2947	778	1729	21,312
Total	**127**	**18,009**	**898**	**10,363**	**173,844**

Рис. 9. Статистика размера кода MINIX 3. Для каждой части показано число файлов, число строк кода на языке С и языке ассемблера, число точек с запятой и размер сегмента текста в байтах.

Интересно, что статистика размеров кода, показанная на рис. 9, представляет минимальную, но функционирующую операционную систему. Общий размер ядерной части и части, работающей в режиме пользователя, составляет всего 18,000 строк кода, необыкновенно мало для POSIX-совместимой операционной системы. Мы сравним эти цифры с другими системами в разд. 6.

6. Родственные исследования

Мы являемся не первыми исследователями, пытающимися предотвратить отказы систем по вине драйверов устройств, содержащих ошибки. И мы не первые пытаемся применить минимальное ядро в качестве возможного решения. Мы даже не являемся первыми среди тех, что реализовывал драйверы, работающие в режиме пользователя. Тем не менее, мы считаем, что мы первыми построили полностью POSIX-совместимую операционную систему с отличными свойствами изоляции сбоев поверх минимального ядра из 3800 строк; в этой системе каждый драйвер выполняется в режиме пользователя в отдельном процессе, а вся ОС выполняется в виде нескольких пользовательских процессов. В этом разделе мы обсудим проекты других исследовательских групп, которые отчасти похожи на то, что делаем мы.

6.1 Изоляция драйверов в программном обеспечении

Одним из важных исследовательских проектов, в котором предпринимается попытка построить надежную систему в присутствии ненадежных драйверов устройств, является Nooks [26]. Целью Nooks является повышение надежности *существующих* операционных систем. Словами авторов, «мы нацеливаем существующие расширения на массовые операционные системы, а не предлагаем новую архитектуру расширений. Мы хотим, чтобы сегодняшние расширения выполнялись на сегодняшних платформах, по возможности, без их изменения.» Идея состоит в обратной совместимости с существующими системами, но небольшие изменения допускаются.

Подход Nooks состоит в том, чтобы оставить драйверы устройств в ядре, но заключить их в своего рода облегченную защитную оболочку, чтобы ошибки драйвера не могли распространяться на другие части операционной системы. Nooks работает путем вставки прозрачного уровня повышения надежности между обертываемым драйверов устройства и оставшейся частью операционной системы. Весь трафик управления и данных между драйвером и оставшейся частью ядра проверяется уровнем повышения надежности. При запуске драйвера уровень повышения надежности модифицирует таблицу страниц ядра таким образом, чтобы запретить доступ по записи к страницам, которые не являются частью драйвера, предотвращая, тем самым, их непосредственную модификацию. Для поддержки законного доступа по записи в структуры данных ядра Nooks копирует необходимые данные в драйвер, а после модификации переписывает их обратно.

Наша цель полностью отличается от цели Nooks. Мы не пытаемся сделать более надежными унаследованные системы. Будучи исследователями, мы задаем вопрос: как следует разрабатывать *будущие* операционные системы, чтобы с самого начала предотвратить возникновение этой проблемы? Мы полагаем, что правильная разработка будущих систем состоит в построении мультисерверной операционной системы и выполнении ненадежного кода в независимых процессах в пользовательском режиме, что сделает этот код гораздо менее вредным (как обсуждалось в разд. 3).

Несмотря на разные цели, имеются и технические аспекты, в отношение которых системы можно сравнивать. Рассмотрим всего несколько примеров. Nooks не может справиться со сложными ошибками, такими как непреднамеренное изменение в драйвере таблицы страниц; в нашей системе у драйверов отсутствует доступ к таблице страниц. Nooks не может справиться с бесконечными циклами; мы можем, поскольку, когда драйвер не отвечает правильным образом серверу реинкарнации, он принудительно завершается и перезапускается. Хотя на практике Nooks может в большинстве случаев справиться с недопустимыми записями в структуры данных ядра, в нашей разработке такие записи не допускаются структурно. Nooks не может справиться с драйвером принтера, который случайно пытается произвести запись в порты ввода-вывода, управляющие диском; мы отлавливаем 100% таких попыток. Заслуживает внимания и размер кода. Nooks включает 22,000 строк кода, почти в шесть раз больше размера всего нашего ядра и больше минимальной конфигурации всей нашей операционной системы. Трудно отойти от этой аксиомы: в большем по размеру коде содержится больше ошибок. Поэтому статистически в Nooks, вероятно, содержится в пять раз больше ошибок, чем во всем нашем ядре.

6.2 Изоляция драйверов с использованием виртуальных машин

В другом проекте по инкапсуляции драйверов это делается с использованием понятия виртуальной машины для их изоляции от других частей системы [19, 18]. Когда драйвер вызывается, он запускается на другой виртуальной машине, не в той, в которой работает основная система, так что его сбой не портит основную систему. Подобно Nooks, этот подход полностью фокусируется на выполнении унаследованных драйверов для унаследованных операционных систем. Авторы не утверждают, что для новых разработок хорошим подходом является включение ненадежного кода в ядро с последующей защитой каждого драйвера путем

его выполнения на отдельной виртуальной машине.

Хотя этот подход позволяет достичь намеченных целей, с ним связаны некоторые проблемы. Во-первых, имеются вопросы, связанные с тем, насколько могут доверять друг другу основная система и виртуальная машина, на которой выполняется драйвер. Во-вторых, запуск драйвера на виртуальной машине порождает проблемы с временными соотношениями и блокировками, поскольку все виртуальные машины работают в режиме разделения времени, и ядерный драйвер, разрабатывавшийся в расчете на выполнение без прерываний, может быть непредвиденным образом квантован во времени с непредусмотренными последствиями. В третьих, может потребоваться совместное использование несколькими виртуальными машинами некоторых ресурсов, таких как конфигурационное пространство шины PCI. В четвертых, механизм виртуальной машины потребляет дополнительные ресурсы, хотя соответствующие расходы соизмеримы с расходами нашей схемы: от 3% до 8%. Хотя для этих проблем предлагаются решения, подход в лучшем случае является громоздким и в основном подходит для защиты унаследованных драйверов в унаследованных операционных системах, а не для использования в новых разработках, которым посвящено наше исследование.

6.3 Средства безопасности, основанные на языках

В предыдущей работе один из авторов также затрагивал проблему безопасного выполнения внешнего кода внутри ядра. В проекте Open Kernel Environment (OKE) обеспечивается безопасная, контролирующая ресурсы среда, позволяющая загрузить в ядро операционной системы Linux полностью оптимизированный собственный код [4]. Код компилируется с использованием специального компилятора Cyclone, который добавляет к объектному коду инструментарий в соответствии с политикой, определяемой привилегиями пользователя. Cyclone, подобно Java, является языком с типовой безопасностью, в котором большая часть ошибок, связанных с указателями, предотвращается языковыми средствами. Явное доверительное управление (trust management) и контроль авторизации обеспечивают администраторам возможность осуществлять строгий контроль над предоставлением внешним модулям привилегий, и этот контроль автоматически приводится в исполнение в коде этих модулей. Кроме обеспечения авторизации, компилятор играет центральную роль в проверке того, что код соответствует установленной политике. Для этого используются как статические проверки, так и динамический инструментарий.

OKE позволяет внешним модулям интенсивно взаимодействовать с другими частями ядра, например, путем совместного использования памяти ядра. Рабочая среда обеспечивает ключевые средства безопасности. В частности, для данных всегда производится сборка мусора, и не может произойти обращение по указателю к свободной памяти. Более того, OKE может обеспечивать контроль надо всеми ресурсами внешних модулей ядра: время ЦП, куча, стек, точки входа и т.д.

Среда OKE разрабатывалась в расчете на написание драйверов и расширений ядра. Однако, поскольку для обеспечения безопасного программирования в ядре Linux требуются процедуры строго контроля доступа и сложные средства, среду довольно трудно использовать. Как отмечают авторы, основная причина состоит в том, что организация Linux просто не предназначена для обеспечения возможности безопасных расширений.

6.4 Виртуальные машины и экзоядра

Классические виртуальные машины [24] представляют собой мощное средство для одновременного выполнения нескольких операционных систем. Экзоядра [10] похожи на виртуальные машины, но в них ресурсы скорее разделяются, а не реплицируются, что приводит к большей эффективности. Однако ни один из этих подходов не решает проблему, поставленную в разд. 1.3: как предотвратить отказы операционных систем по вине драйверов устройств,

содержащих ошибки?

6.5 Драйверы, выполняющиеся в пользовательском режиме в монолитном ядре

Ранним проектом, в котором применялись драйверы, выполняющиеся в пользовательском режиме, был Mach 3.0 [11]. Система состояла из микроядра Mach, поверх которого запускалась ОС Berkeley UNIX в виде пользовательского процесса, и драйверы устройств также выполнялись в пользовательских процессах. К сожалению, в случае фатального сбоя драйвера Berkeley UNIX приходилось перезапускать, так что от изоляции драйверов было мало пользы. Планировалась мультисерверная система, которая должна была выполняться над Mach, но она так и не была полностью реализована.

В аналогичном проекте в университете New South Wales реализовывались драйверы Linux для жесткого диска и гигабайтной аппаратуры Ethernet, выполняемые в пользовательском режиме [8]. Для блоков размером менее 32 Кб производительность ядерного драйвера была значительно выше, но на блоках большего размера выравнивалась. При тестировании Ethernet выявилось так много аномалий, вероятно, связанных с управлением буферами, но невозможно было сделать какие-либо выводы.

6.6 Разработки минимальных ядер

Хотя извлечение драйверов из ядра является большим шагом вперед, еще лучше извлечь из ядра операционную систему. Именно здесь начинают применяться минимальные ядра с чрезвычайным сокращением числа реализуемых в них абстракций. Вероятно, первым минимальным ядром была система RC4000 Бринка Хансена (Brinch Hansen), датируемая началом 1970-х гг. [13]. С середины 1980-х гг. был написан ряд минимальных ядер, включая Amoeba [21], Chorus [5], Mach [1] и V [6]. Однако ни в одном из них не применялось безопасное программное обеспечение: у всех имелись не изолированные драйверы внутри ядра.

QNX является коммерческой UNIX-подобной системой реального времени с закрытыми кодами [17]. Хотя у нее имеется минимальное ядро, называемое Neutrino, по поводу системы опубликовано мало статей, и точные детали нам неизвестны. Однако на основе последних проспектов мы заключаем, что Neutrino является гибридным ядром, поскольку менеджер процессов работает в адресном пространстве ядра.

В начале 1990 гг. покойный Йохан Лидтке (Jochen Liedtke) написал минимальное ядро L4 на языке ассемблера для архитектуры x86. Быстро стало понятно, что оно не является переносимым, и его трудно поддерживать, и поэтому он переписал ядро на языке C [20]. После этого оно продолжало развиваться. В настоящее время имеются две основные ветви: L4/Fiasco, поддерживаемое в техническом университете Дрездена, и L4Ka::Pistachio, поддерживаемое в университете Карлсруэ и университете New South Wales. Они написаны на C++.

Ключевыми идеями в L4 являются адресные пространства, нити и IPC между нитями в разных адресных пространствах. Менеджер ресурсов, выполняемый в пользовательском режиме и запускаемый при загрузке системы, управляет системными ресурсами и распределяет их между пользовательскими процессами. L4 – это одно из немногих действительно минимальных ядер с драйверами устройств, работающими в пользовательском режиме. Однако отсутствует реализация, в которой каждый драйвер выполнялся бы в отдельном адресном пространстве, и API L4 совсем отличается от нашего API, поэтому мы не можем запустить на нем какие-либо тесты.

Однако оказалось нетрудно запустить скрипт подсчета числа строк над текущей версией ядра L4Ka::Pistachio. Результаты показаны на рис. 10, и их можно сравнить с данными на строке «Kernal» рис. 9. Размер исходного кода почти в два раза превышает размер нашего ядра, а

бинарный код в шесть раз больше, однако функциональные возможности L4Ka::Pistachio являются совсем другими, так что трудно сказать что-нибудь еще, кроме того, что это ядро значительно больше по размеру.

Part	Files	C++	Asm	;	Binary
Kernel	57	6881	420	3053	114 KB

Рис. 10. Статистика размера кода для L4Ka::Pistachio.

6.7 Односерверные операционные системы

Одним из способов использования минимальных ядер является обеспечение платформы, поверх которой, как единственный сервер, запускается вся операционная система, возможно, в режиме пользователя. Для получения системных сервисов пользовательские программы запрашивают их у процесса операционной системы. Свойства такой архитектуры аналогичны свойствам монолитных систем, обсуждавшимся в разд. 2.1. Ошибка в драйвере по-прежнему может сломать всю операционную систему, а в результате и прикладные программы. Поэтому, с точки зрения изоляции сбоев, выполнение всей операционной системы в одном пользовательском процессе ничуть не лучше ее выполнения в режиме ядра. Единственным реальным преимуществом является то, что перезагрузка после фатального сбоя сервера операционной системы, выполняемого в режиме пользователя, и всех приложений происходит быстрее, чем перезагрузка компьютера.

Одним из примеров этой технологии является ОС Berkeley UNIX поверх Mach (переименованная в Darwin компанией Apple), которая является основой системы Apple Mac OS X [28]. Однако в этой системе UNIX выполняется в ядре, что делает его просто иначе структурированным монолитным ядром. Второй пример – ОС MkLinux, в которой Linux выполняется в единственном пользовательском процессе поверх Mach. Третий пример – L4-Linux, в которой полный вариант Linux выполняется поверх L4 [15]. В последней из перечисленных систем пользовательские процессы получают сервисы операционной системы путем вызова удаленных процедур в сервере Linux с использованием механизма IPC L4. Измерения показывают падение производительности по сравнению с обычной ОС Linux на 5-10%, что очень близко к нашим наблюдениям. Однако единственная строка с ошибочным кодом в драйвере Linux может привести к фатальному сбою всей операционной системы, так что единственным преимуществом этой архитектуры с точки зрения надежности является более быстрая загрузка.

6.8 Мультисерверные операционные системы

Более сложный подход состоит в расщеплении операционной системы на части и выполнении каждой части в собственной области защиты. Одним из таких проектов был SawMill Linux [12]. Однако в 2001 г. проект был неожиданно остановлен после того, как многие из его основных участников ушли из IBM.

Другим мультисерверным проектом является DROPS, в котором ОС также строится поверх минимального ядра L4/Fiasco [14]. Этот проект ориентирована на мультимедийные приложения. Однако большинство драйверов устройств выполняется в составе большого серверного процесса L4-Linux, и только мультимедийные подсистемы выполняются отдельно. После некоторой настройки проигрыш в производительности снизился до 2-4%.

Еще одной мультисерверной операционной системой с драйверами, выполняемыми в пользовательском режиме, является Nemesis [23]. В этой системе имеется единое адресное пространство, разделяемое всеми процессами, но используется аппаратная защита между процессами. Подобно DROPS эта система была ориентирована на мультимедийные приложения,

но не являлась POSIX-совместимой и даже UNIX-подобной.

7. Заключение

Основное достижение работы, описанной в этой статье, состоит в том, что мы построили POSIX-совместимую операционную систему, основанную на минимальном ядре, исходные тексты которого составляют менее 3800 строк. Только этот код выполняется в режиме ядра. Насколько нам известно, наше минимальное ядро является наименьшим среди всех существующих ядер, которые поддерживают полностью POSIX-совместимую мультисерверную операционную систему, функционирующую в пользовательском режиме. Уникальность нашей системы состоит также в том, что в ней каждый драйвер устройства выполняется в отдельном пользовательском процессе, и имеется возможность реинкарнации бездействующих или неверно функционирующих драйверов на лету, без перезагрузки операционной системы. Мы не утверждаем, что можем отловить любую ошибку, но мы существенно повысили надежность операционной системы путем структурного устранения многих различных классов ошибок.

Для достижения максимальной надежности в своей разработке мы руководствовались принципами простоты, модульности, наименьшей авторизации и отказоустойчивости. В понимаемом и минимальном ядре содержится меньшее число ошибок, и оно в меньшей степени подвержено фатальным сбоям. Например, в нашем коде ядра невозможны переполнения буферов, поскольку все структуры данных в нем объявляются статически, а не с использованием динамического распределения памяти. Кроме того, путем перемещения большей части кода (и большей части ошибок) в непривилегированные пользовательские процессы и ограничения возможностей каждого из них мы добились должной изоляции сбоев и ограничили масштаб соответствующего потенциального ущерба. Более того, большинство серверов и все драйверы в операционной системе подвергаются мониторингу и автоматически восстанавливаются при обнаружении проблемы. За это сокращение числа фатальных сбоев операционной системы мы платим снижением производительности на 5-10%. Мы считаем эту цену вполне обоснованной.

Конечно, драйверы, файловые системы и другие компоненты не становятся в нашей разработке магическим образом безошибочными. Однако при наличии стабильного минимального ядра сценарий *наихудшего случая* изменяется от потребности в перезагрузке компьютера к потребности в перезапуске операционной системы в режиме пользователя. По крайней мере, это восстановление происходит гораздо быстрее. В лучшем случае, если, скажем, в драйвере принтера возникает аварийный отказ по причине записи по неверному указателю, сервер реинкарнации автоматически запускает свежую копию этого драйвера. Потребуется заново выполнить текущее задание на печать, но все это никак не повлияет на другие программы, которые выполнялись к моменту фатального сбоя драйвера. Ситуация с блочными устройствами обстоит еще лучше. Если обнаруживается сбой дискового драйвера, то система может произвести полное восстановление путем прозрачной замены драйвера и перезаписи блоков из буферного кэша файловой системы.

В завершение статьи заметим, что мы показали, как можно повысить надежность операционной системы с использованием элегантного, облегченного подхода. Наша система в настоящее время является устойчивой к большинству видов неверной работы, вызываемой ошибками. Однако имеются новые проблемы, связанные со злоумышленными серверами и драйверами. Мы продолжаем исследовательскую работу в этой области.

8. Благодарности

Мы хотели бы поблагодарить Бена Граса (Ben Gras) и Филиппа Хомбурга (Philip Homburg) за выполнение тестовых программ, некоторое участие в программировании и внимательное чтение статьи. Мы также благодарны Яну Луену, Руеди Вейсу, Бруно Крипсо и Кэрол Конти за их

помощь и замечания.

9. Литература

[1] M. Acceta, R. Baron, W. Bolosky, D. Holub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. In *Proc. 1986 USENIX Summer Tech. Conf.*, pages 93–112, June 1986. http://www.cs.toronto.edu/~demke/469F.06/Handouts/mach_usenix86.pdf

[2] V. Basili and B. Perricone. Software Errors and Complexity: An Empirical Investigation. *Commun. of the ACM*, 21(1):42–52, Jan. 1984. http://www.cs.umd.edu/~basili/publications/journals/J20.pdf

[3] B. Bershad. The Increasing Irrelevance of IPC Performance for Microkernel-Based Operating

Systems. In *Proc. Usenix Microkernels Workshop*, pages 205–211, Apr. 1992. http://citeseer.ist.psu.edu/bershad92increasing.html

[4] H. Bos and B. Samwel. Safe Kernel Programming in the OKE. In *Proc. of the 5th IEEE Conference on Open Architectures and Network Programming*, pages 141–152, June 2002. http://www.ist-scampi.org/publications/papers/bos-openk.pdf

[5] A. Bricker, M. Gien, M. Guillemont, J. Lipkis, D. Orr, and M. Rozier. A New Look at Microkernel-Based UNIX Operating Systems: Lessons in Performance and Compatibility. In *Proc. EurOpen Spring 1991 Conf.*, pages 13–32, May 1991. http://citeseer.ist.psu.edu/bricker91new.html

[6] D. Cheriton. The V Kernel: A Software Base for Distributed Systems. *IEEE Software*, 1(2):19–42, Apr. 1984. http://www.cs.ucsb.edu/~ravenben/papers/coreos/Che88.pdf

[7] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An Empirical Study of Operating System Errors. In *Proc. 18th ACM Symp. on Oper. Syst. Prin.*, pages 73–88, 2001. http://pdos.csail.mit.edu/6.097/readings/osbugs.pdf

[8] P. Chubb. Get More Device Drivers Out of the Kernel! In *Proc. Linux Symp.*, pages 149–162, July 2004. http://www.ertos.nicta.com.au/publications/papers/Chubb_04b.pdf

[9] Сент-Экзюпери А. де Соч.: В 3 т. - Рига: Полярис, 1997. - т.1, с.179-308.

[10] D. Engler, M. Kaashoek, and J. J. O'Toole. Exokernel: an operating system architecture for application-level resource management. In *Proc. 15th ACM Symp. on Oper. Syst. Prin.*, pages 251–266, 1995. http://www.di.unipi.it/~scordino/sisop/Exokernel.pdf

[11] A. Forin, D. Golub, and B. Bershad. An I/O System for Mach 3.0. In *Proc. Second USENIX Mach Symp.*, pages 163–176, 1991. http://citeseer.ist.psu.edu/45179.html

[12] A. Gefflaut, T. Jaeger, Y. Park, J. Liedtke, K. Elphinstone, V. Uhlig, J. Tidswell, L. Deller, and L. Reuther. The SawMill Multiserver Approach. In *ACM SIGOPS European Workshop*, pages 109–114, Sept. 2000. http://i30www.ira.uka.de/research/documents/l4ka/sawmill-multiserver.pdf

[13] P. B. Hansen. *Operating System Principles*. Prentice Hall, 1973. http://brinch-hansen.net/papers/1969c.pdf

[14] H. Hartig, R. Baumgartl, M. Borriss, C.-J. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schonberg, and J. Wolter. DROPS OS Support for Distributed Multimedia Applications. In *Proc. 8th ACM SIGOPS European Workshop*, pages 203–209, Sept. 1998. http://www.tudos.org/papers_ps/sintra98.ps

[15] H. Hartig, M. Hohmuth, J. Liedtke, S. Schonberg, and J. Wolter. The Performance of μ-Kernel-Based Systems. In *Proc. 16th ACM Symp. on Oper. Syst. Prin.*, pages 66–77, Oct. 1997. http://i30www.ira.uka.de/research/documents/l4ka/ukernel-performance.pdf

[16] H. Hartig, J. Loser, F. Mehnert, L. Reuther, M. Pohlack, and A. Warg. An I/O Architecture for Microkernel-Based Operating Systems, July 2003. Technical Report. TU Dresden. http://os.inf.tu-dresden.de/papers_ps/tr-ioarch-2003.pdf

[17] D. Hildebrand. An Architectural Overview of QNX. In *Proc. USENIX Workshop in Microkernels and Other Kernel Architectures*, pages 113–126, Apr. 1992. http://ftp.funet.fi/pub/OS/QNX/doc/qnx-paper.ps.Z

[18] J. LeVasseur and V. Uhlig. A Sledgehammer Approach to Reuse of Legacy Device Drivers. In *Proc. 11th ACM SIGOPS European Workshop*, pages 131–136, Sept. 2004. http://l4ka.org/publications/2004/levasseur04sledgehammer.pdf

[19] J. LeVasseur, V. Uhlig, J. Stoess, and S. Gotz. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *Proc.*

6th Symp. on Oper. Syst. Design and Impl., pages 17–30, Dec. 2004. http://i30www.ira.uka.de/research/documents/l4ka/2004/LeVasseur04UnmodifiedDriverReuse.pdf

[20] J. Liedtke. On μ-Kernel Construction. In *Proc. 15th ACM Symp. on Oper. Syst. Prin.*, pages 237–250, Dec. 1995. http://citeseer.ist.psu.edu/liedtke95microkernel.html

[21] S. Mullender, G. V. Rossum, A. Tanenbaum, R. V. Renesse, and H. V. Staveren. Amoeba: A Distributed Operating System for the 1990s. In *IEEE Computer Magazine 23(5)*, pages 44–54, May 1990. http://www.cs.cornell.edu/Info/People/rvr/papers/Amoeba1990s.pdf

[22] T. Ostrand, E. Weyuker, , and R. Bell. Where the Bugs Are. In *Proc. of the 2004 ACM SIGSOFT Int'l Symp. on Software Testing and Analysis*, pages 86–96. ACM, 2004.

[23] T. Roscoe. The Structure of a MultiService Operating System. Ph.D. Dissertation, Cambridge University. http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-376.ps.gz

[24] L. Seawright and R. MacKinnon. VM/370—A Study of Multiplicity and Usefulness. *IBM Systems Journal*, 18(1):4–17, 1979. http://www.cs.wisc.edu/~stjones/proj/vm_reading/ibmsj1801C.pdf

[25] M. Swift, M. Annamalai, B. Bershad, and H. Levy. Recovering Device Drivers. In *Proc. Sixth Symp. on Oper. Syst. Design and Impl.*, pages 1–15, 2004. http://nooks.cs.washington.edu/recovering-drivers.pdf

[26] M. Swift, B. Bershad, and H. Levy. Improving the Reliability of Commodity Operating Systems. 23(1):77–110, 2005. http://nooks.cs.washington.edu/nooks-tocs.pdf

[27] T.J. Ostrand and E.J. Weyuker. The Distribution of Faults in a Large Industrial Software System. In *Proc. of the 2002 ACM SIGSOFT Int'l Symp. on Software Testing and Analysis*, pages 55–64. ACM, 2002.

[28] A. Weiss. Strange Bedfellows. *netWorker*, 5(2):19–25, June 2001.