МАТЕМАТИЧЕСКИЕ ОСНОВЫ ИНФОРМАТИКИ И ПРОГРАММИРОВАНИЯ

DOI 10.17223/20710410/10/3

УДК 681.3

МОДЕЛЬ ЗАЩИЩЁННОЙ АРХИТЕКТУРЫ ЭВМ И ЕЁ ВИРТУАЛЬНАЯ РЕАЛИЗАЦИЯ[1]

В. В. Горелов

Национальный исследовательский Томский государственный университет, г. Томск, Россия

E-mail: skylark@mail.tsu.ru

Предлагается модель защищённой вычислительной архитектуры со сквозной защитой типов данных и ссылок. Сквозная защита достигается за счёт контроля границ значений типов данных и ссылок и допустимых операций для них. Такой подход делает возможной единообразную защиту ресурсов на всех уровнях: машинных инструкций, прикладных программ, операционной системы. Текущая реализация данной архитектуры произведена через разработку виртуальной машины, работающей поверх существующей операционной системы общего назначения.

Ключевые слова: *программное обеспечение, вычислительная архитектура, безопасность, виртуализация, изоляция ошибок.*

Введение

На протяжении последних десятилетий развитие массовых электронных вычислительных систем общего назначения происходит скачкообразными темпами. Если аппаратная составляющая развивается в основном экстенсивным методом (увеличение объёма оперативной и постоянной памяти, разрядности процессора, частоты и количества ядер), то архитектурная составляющая меняется значительно медленнее. Пожалуй, единственным серьёзным архитектурным изменением, с точки зрения безопасности, в архитектуре процессоров (Intel-совместимых и с аналогичной архитектурой) явилось использование защищённого режима процессора. В данном режиме адресное пространство процесса и ядра ОС защищено от случайного ошибочного или умышленного злонамеренного поведения другого процесса. Компоненты вычислительной системы могут находиться в защищённых друг от друга областях памяти. Однако на этом изоляция адресных пространств заканчивается. Внутри этих разделённых областей никакой защиты нет. Ни ядро операционной системы, ни прикладные программы не защищены внутри себя от непреднамеренных ошибок или умышленного «склонения» поведения кода в нужную злоумышленнику сторону.

С целью повышения надёжности и безопасности разработчики современных языков программирования (Ада, Ява, Си#) предусмотрели ряд мер, предотвращающих ненадёжное и небезопасное поведение программ, однако все они направлены исключительно на программы, написанные на этих языках, а не на вычислительную архитектуру

[1]Работа выполнена в рамках реализации ФЦП «Научные и научно-педагогические кадры инновационной России» на 2009–2013 гг. (гос. контракт № П1010).

в целом. Кроме того, для обеспечения проверки целостности типов и указателей в код внедряются программные проверки разнообразных границ, что неизбежным образом замедляет работу программ. В двух последних языках программирования обязательным компонентом является сборщик мусора, лишающий программу важной характеристики: работа в реальном времени. Данный режим работы гарантирует максимальное время реакции компонентов вычислительной системы на определённое событие, что при работе сборщика мусора не представляется возможным. Последние два языка также изначально спроектированы как языки прикладные, т. е. написание на них кода ядра операционной системы крайне затруднительно, если вообще возможно.

Язык программирования существующих широко распространённых ядер — Си. Если часть ядра, которая не зависит от внешних устройств, относительно проработана, то большое количество плат и устройств расширений ЭВМ требует драйверов, которые динамически подключаются в адресное пространство ядра для обеспечения связи между ядром и «железом». Драйверы зачастую разрабатываются сторонними (не основными) разработчиками ядра. Это влияет как на количество разнообразных ошибок, так и на сам формат драйвера, которым они снабжают пользователей своих устройств. Часть драйверов поставляются без исходных текстов, т. е. в машинном виде, что делает их анализ, нахождение и устранение ошибок максимально трудоёмким. Самостоятельная разработка драйверов на системном языке Си не всегда возможна ввиду того, что производитель не раскрывает спецификацию протокола взаимодействия со своим устройством. В частности, производители видеокарт не открывают исходные коды своих драйверов ещё и по причине наличия в них сомнительных приёмов для:

— подгонки производительности под конкретное ПО (для тестов производительности) с целью получения видимости конкурентного преимущества перед аналогичными показателями тестов видеокарт других производителей;
— трюков, позволяющих повысить производительность за счёт качества изображения;
— сокрытия механизмов постепенного ухудшения производительности и функциональности «старых» видеокарт.

Подключение таких закрытых[2] производителем модулей в настоящее время почти всегда является единственной возможностью использовать устройство, но в то же время позволяет работать на уровне ядра ошибочным и потенциально злонамеренным драйверам. Недоступность исходного кода драйверов, кроме вышеописанного, влечёт за собой затруднённое обнаружение не только ошибок внутри самого драйвера, но и его пагубного воздействия на другие компоненты ядра.

Текущее положение вещей представляется весьма неблагоприятным для надёжности и безопасности как отдельных компонентов вычислительной системы, так и для всей системы в целом. Далее предлагается экспериментальная модель архитектуры (называемая «композит») со сквозной защитой данных и ссылок, достигаемой путём контроля границ значений типов данных и ссылок, а также допустимых операций для них. Тот факт, что такой контроль заложен непосредственно в архитектуре, даёт возможность работы всей вычислительной системы с высокой степенью изоляции от внешних и внутренних ошибок на всех уровнях: машинные коды, ядро операционной системы, прикладные программы, т. е. защита распростаняется на все компоненты: от самого низкого уровня «доступа к железу» до уровня прикладных программ. Це-

[2]Продолжая Л. Н. Гумилёва: «А там, где тайна, там дьявол, ложь, гордыня и злоба».

лью архитектуры является выявление и, как следствие, ликвидация широкого класса дефектов в программном обеспечении.

Статья состоит из двух частей. В первой части (п. 1 – 6) описана модель защищённой архитектуры ЭВМ. Представлены типы данных архитектуры, операции над ними, вычислительные модули, смена контекста выполнения вычислительных модулей и поддержка исключений. Во второй части (п. 7 – 11) описаны разные подходы виртуализации, процессы загрузки и выполнения ОС для защищённой архитектуры, а также текущее состояние реализации и перспективы разработки.

1. Типы данных архитектуры

Типы данных делятся на два принципиально различных класса:

1) непосредственно данные;
2) ссылки на данные.

Первые представляют собой величины для хранения и вычисления целых и вещественных значений. Размерность этих базовых значений может быть 1, 2, 4 или 8 байт для целых и 4 или 8 байт для вещественных типов.

1.1. Целые типы данных архитектуры

Целые делятся на два класса:

1) с представлением неотрицательных чисел;
2) с представлением отрицательных и неотрицательных чисел.

Допустимый диапазон значений для первого класса $[0; 2^n - 1]$; для второго — $[-2^{n-1}; 2^{n-1} - 1]$, где n — разрядность в битах.

В свою очередь, каждый из классов целых чисел разделён на два других: тип по модулю и без модуля. Непосредственно поддерживаются следующие типы переменных: `uchar`, `ushort`, `uint`, `schar`, `sshort`, `sint`, `mod8`, `mod16`, `mod32`. Первые три — беззнаковые типы с размерностью 8, 16, 32 бит соответственно; за ними следуют знаковые типы тех же размерностей и закрывают перечисление модульные типы.

Для типов по модулю операции над ними проводятся по модулю 2^8, 2^{16} или 2^{32} соответственно. Для безмодульных типов операции, приводящие к результату, лежащему за пределами допустимого диапазона, вызывают диагностическое прерывание.

1.2. Подтипы

Допускается введение подтипов для базовых типов данных. Подтипы — новые типы с суженными для них границами диапазонов допустимых значений. В новом подтипе допустимы такие же операции, как и для родительского типа, но с учётом ограничений подтипа.

В операциях присваивания родительскому типу значения его подтипа не требуется указание типа, так как заранее известно, что его диапазон шире, чем у подтипа. Обратное не верно, операция указания типа необходима, и в случае нарушения допустимого диапазона подтипа вызывается диагностическое прерывание.

1.3. Теги

Каждая переменная, кроме типа данных и самих данных, снабжена тегом, указывающим допустимые для неё операции — чтение и/или запись. Поскольку теги и другие вспомогательные информационные блоки необходимы только для архитектуры, то видны они только «процессору» архитектуры, оставаясь недоступными на уровне кода программы. Платой за это является бóльший расход физической памяти.

2. Операции

Представленные здесь операции являются лишь минимальной основой для работы разработанного прототипа [1] виртуальной машины, реализующей архитектуру. С развитием архитектуры и более широком её использовании на практике их множество, очевидно, будет расширяться. Коды основных операций следующие:

— `op_call`: вызов функции;
— `op_jmp`: переход к именованной метке;
— `op_ret`: возврат из функции;
— `op_nop`: пустая операция;
— `op_mov`: операция пересылки;
— `op_add`: операция сложения;
— `op_sub`: операция вычитания;
— `op_mul`: операция умножения;
— `op_div`: операция деления;
— `op_out`: операция печати (на «принтер» — фактически байты выводятся на стандартный поток вывода эмулятора).

Каждая операция состоит из:

— номера строки (необходим при выводе диагностической информации);
— метки (необязательная метка для перехода);
— кода операции;
— списка аргументов.

3. Вычислительный модуль

Этот модуль, называемый иногда сокращённо ВМ, представляет собой базовый блок архитектуры, защищённый от недружественного воздействия извне. Он имеет доступ непосредственно к переменным и ссылкам, переданным ему как параметры, и больше ни к чему доступа не имеет. Создать произвольные ссылки, указывающие на не принадлежащие ему величины, он также не может.

Описательная часть модуля состоит из:

— имени;
— списка входных типов и их символьного обозначения;
— списка собственных переменных, доступных для оперирования;
— списка собственных массивов переменных;
— списка операций.

4. Текущий контекст

Текущий контекст задаёт состояние вычислительного модуля с переданными ему аргументами, с его собственными аргументами и с текущим курсором порядкового номера исполняемой операции. Номер используется исключительно процессором и не доступен для выполняющегося кода.

5. Поддержка исключений

Архитектура предусматривает поддержку исключений как один из базовых механизмов по обнаружению и реагированию на внештатные ситуации. В рамках вычислительного модуля предусматривается возможность создания блоков обработки исключительных ситуаций. Для обработки исключительной ситуации в конце тела модуля создаются метки вида `:when_overflow`, `:when_outofbounds`, `:when_outofmemory`,

`:when_ioerror` и т. д., в рамках которых алгоритм может должным образом среагировать на возникшую ситуацию и при возможности продолжить работу. В случае отсутствия необходимого обработчика исключения в текущем модуле контекст закрывается и выполняется операция сворачивания контекстов и поиска обработчика. Операция поиска продолжается до тех пор, пока «размотка» контекстов не доберётся до обработчика исключений по умолчанию, который закроет экземпляр ОС. Вся информация об исключении сохраняется для последующего анализа.

Ввиду поддержки исключений представляется возможным исполнение программ, написанных как на низкоуровневых языках программирования (Си и т. п.), так и на высокоуровневых (Ада, Ява, Си# и т. п.).

6. Поддержка языков программирования

Для поддержки существующего программного обеспечения необходима разработка или, скорее всего, адаптация существующих компиляторов путём добавления в них возможности создавать код для защищённой архитектуры. Для достаточно низкоуровневого языка программирования Си (и Си++, как основанного на нём) ожидаемой будет ситуация, подобная следующей. Предположим, что где-то в исходном коде есть некая переменная i типа `unsigned int`, которая изменяется во время исполнения программы. Если значение переменной максимально $(2^{32} - 1)$ и к нему прибавляется единица, возможны два случая.

— Счётчик переполнен. В этом случае далее алгоритм будет работать неверно, так как значение счётчика станет равным 0 и выражение $(i + 1 > i)$ будет ложно. Поскольку с точки зрения процессора архитектуры произошло нарушение границ типов, задача будет остановлена с протоколированием стека для последующего разбирательства.

— Возможен и другой случай, когда по логике алгоритма так задумано умышленно: прибавление 1 к максимальному допустимому значению должно выполняться по модулю 2 в степени (размер счётчика в битах), и аварийный останов программы недопустим. Автоматически отличить этот случай от первого не представляется возможным, поэтому для формального их разделения необходима более точная конкретизация языка программирования, например через введение в него типа `mod int`.

При применении адаптированного компилятора для платформы будут получены два очевидных плюса:

— автоматическое обнаружение ошибочных и потенциально ошибочных мест в программах;

— возможность запуска на данной архитектуре существующего программного обеспечения.

7. Системы виртуализации

7.1. Типы виртуализации

Современные системы виртуализации весьма обширны по своей природе и чаще всего бывают следующих типов:

1) Полная виртуализация архитектуры. Для запускаемой ОС создаётся впечатление, что она работает поверх реального «железа», и для ОС зачастую нет никакой возможности понять, что она виртуализирована. Данный тип подходит для виртуализации практически всего, что работает на эмуляторе соответствующей

архитектуры. Обычно различают архитектуру машины-хозяина (Host) и архитектуру машины-гостя (Guest). Под машиной-хозяином имеется в виду главная (физическая) машина, на которой запущено средство виртуализации, а под гостевой — система, запущенная внутри средства виртуализации. Их архитектуры могут не совпадать. Например, в эмуляторе, работающем в ОС на архитектуре x86, можно запустить ОС для архитектуры PowerPC и наоборот. Типичным представителем данного типа виртуализации является проект QEMU [2]. Этот тип виртуализации низкоскоростной, однако существуют различные эффективные оптимизации, значительно ускоряющие процесс. Проникновение злоумышленного кода из гостевой системы в систему хозяина маловероятно.

2) Частичная виртуализация. В этом случае, как правило, архитектуры гостевой и хозяйской систем совпадают. Часть гостевых команд запускается прямо на процессоре хозяина, другая часть (привилегированные команды) динамически преобразуется в этот тип. Хорошо подходит для виртуализации произвольных ОС на данной архитектуре. Отличается довольно высокой скоростью работы. Типичные представители этого типа — VMWare [3], VirtualBox [4]. Проникновение злоумышленного кода из гостевой системы в систему хозяина маловероятно.
3) Паравиртуализация. Ядро гостевой системы должно быть определённым образом подготовлено для такой работы. Подготовка несколько напоминает перенос на другую платформу. Заключается это в перенастройке ядра на использование специализированных обращений к гипервизору Xen для работы в привилегированном состоянии процессора, для захвата и управления физической памятью, устройствами ввода-вывода. Этот тип отлично подходит для ядер с открытым кодом. Данная технология успешно применена для Linux [5], FreeBSD [6]. Для закрытых систем требуется модификация с разрешения компании-владельца. Запуск таких систем возможен и без их модификации, однако в данном случае выигрыш в быстродействии не наблюдается. Типичным представителем этого направления является технология Xen [7]. Паравиртуализация отличается ещё бóльшим быстродействием. Проникновение злоумышленного кода из гостевой системы в систему хозяина зависит от многочисленных факторов.
4) Виртуализация уровня ОС. В данном подходе изменяется ОС машины-хозяина, чтобы создать внутри неё изолированное окружение. В этом случае скорость работы программ ещё выше, поскольку инструкции работают напрямую — поверх самой ОС. Архитектуры гостя и хозяина совпадают. Защита осуществляется силами самой ОС. Взлом таких систем наиболее вероятен. Типичные представители этого: Solaris Zones [8], FreeBSD jail [9], OpenVZ [10], chroot [11].

7.2. Взаимовлияние хозяина и гостевых систем

Общим свойством описанных выше систем виртуализации является тот факт, что для полноценной работы гостевой системы в неё необходимо добавление драйверов или других изменений ядра, реализующих каналы информационного обмена с хозяином. С точки зрения производительности это безусловный плюс. С точки зрения безопасности появляется возможность совершить поднятие полномочий с гостевых до уровня хозяина. Кроме этого, если гостевые ОС более-менее изолированы друг от друга, то главная ОС имеет полный доступ к любой из гостевых систем.

В разрабатываемой архитектуре изначально предполагается сделать реальным и естественным безопасное и эффективное сосуществование разных ОС для данной архитектуры. Механизмом для этого служит системный монитор. В его основные задачи

входит запуск, остановка и переключение между запущенными ОС. Таким образом, все запускаемые ОС изолированы друг от друга и над ними не довлеет хозяйская система.

7.3. Дополнительные плюсы виртуализации

Кроме прямых плюсов виртуализации — возможности запускать несколько ОС на одной физической ЭВМ, у этой технологии есть и относительно недавно открытая возможность миграции гостевых систем между системами хозяев. Это находит широкое применение при обновлении аппаратного обеспечения, когда в режиме реального времени практически без прерывания работы гостевая ОС «переезжает» на новую машину. Кроме этого, такой механизм зачастую применяется при перераспределении нагрузки в вычислительных центрах и для повышения отказоустойчивости систем. Такие технологии входят в так называемые облачные вычисления на стороне вычислительных центров.

8. Виртуальная ОС защищённой архитектуры

Она состоит из набора вычислительных модулей, векторов для оперирования с контекстами вычислительных модулей, индекса текущего контекста.

Экспериментальный вариант виртуальной машины, реализующей защищённую архитектуру, реализован для загрузки мнемонического представления программ на этапе загрузки. Далее он преобразуется во внутренние структуры процессора. Часть проверок, которые можно отследить при трансляции кода, реализована на этапе загрузки. В аппаратной же реализации проверки при трансляции следует отдать транслятору, а проверки на этапе выполнения — процессору. Ввиду программной реализации архитектуры (в виде эмулятора), все проверки будут описаны без явного разделения на фазу трансляции и фазу проверки во время исполнения кода.

9. Процесс загрузки экземпляра (виртуальной) ОС

ОС представляет собой набор вычислительных модулей. Её загрузка состоит из трёх фаз. До выполнения первой создаётся пустая запись, идентифицирующая очередной загружаемый вычислительный модуль.

9.1. Фаза 1. Загрузка описания (заголовка) вычислительного модуля

В данной фазе производится проверка синтаксиса и семантики описания вычислительного модуля. Каждый модуль должен начинаться с ключевого слова `.name` с указанием имени модуля и следующим за ним списком пар вида (тип, имя). В семантическую составляющую входит проверка существования модуля с таким же именем, корректности названий типов, а также уникальности названий переменных в рамках данного вычислительного модуля. Разрешённые операции для аргументов устанавливаются истинными для записи и чтения, если не определено обратное. Если проверка прошла успешно, то в запись о ВМ вносятся соответствующие поля и осуществляется переход к следующей фазе. В случае нарушения правил синтаксиса или семантики выдаётся сообщение об ошибке и номер её строки.

9.2. Фаза 2. Загрузка локальных переменных вычислительного модуля

В данной фазе также проводится проверка синтаксических и семантических конструкций. Сигнальным элементом начала определения блока переменных является

ключевое слово `.var`. Для определения одной переменной построчно должны быть считаны пары (тип, имя).

Синтаксически верными именами переменных являются имена, начинающиеся с буквы, далее может располагаться произвольная комбинация цифр, букв и символов подчёркивания.

Для неинициализированных переменных допустимость операции чтения устанавливается как ложь, операции записи — как истина. Это не касается случая, когда задаётся константа через элемент `const`, который следует после описания типа. В этом случае допустимость чтения — истина, записи — ложь.

Кроме одиночных переменных, существует понятие массива — адресуемого множества с переменными одного типа. В блоке описания переменных `.var` массивы задаются следующим образом:

```
array TYPE NAME SIZE VALUE
```

или как

```
array TYPE NAME SIZE VALUE1 VALUE2 ...
```

В первом случае ключевое слово `array` указывает, что описывается массив; затем идёт его тип, размер и значение, которым он будет инициализирован. Во втором случае значения `VALUE1`, `VALUE2`, ... будут циклически записаны в элементы массива. Значения для инициализации массива проверяются на границы допустимости для типа массива. Неотъемлемым атрибутом массива является его размер. Все операции по индексированию проверяются на соблюдение границ.

Для задания строк допустима такая форма:

```
array mod8 My_String "Здравствуй, мир!".
```

Данная форма — аналог байтного массива. Размер массива определяется на этапе трансляции. В текущей реализации для простоты все символы кодируются однобайтной кодировкой KOI8-R. Такая форма задания строк допускается как для удобства программиста, так и для повышения читаемости кода.

Так же, как и в предыдущей фазе, любое нарушение синтаксиса или семантики выводится как ошибка с пояснением и номером строки.

9.3. Фаза 3. Загрузка кодов операций вычислительного модуля

Ключевым словом к разбору тела модуля является `.begin`. В данной фазе, как и в двух предыдущих, проводится синтаксическая проверка соответствия названий имён переменных, операций и др. Семантическая проверка сверяет область видимости вызываемых функций, совместимость типов и их количество, передаваемых при вызове других модулей. Все операции, прошедшие проверку, записываются в кортеж операций текущего вычислительного модуля с заполнением всех полей. Для модулей не допускается отсутствие операций. Сигналом окончания текущего модуля является ключевое слово `.end`. После этого текущий вычислительный модуль помещается в кортеж вычислительных модулей для текущей ОС. На этом фаза загрузки вычислительных модулей завершена.

9.4. Ссылки

Ссылку можно создать специальной командой, аргументом которой является имя переменной или массива, на который она должна указывать. При создании ссылки на массив необязательным, но полезным является параметр, конкретизирующий границы, доступные для индексации через создаваемую ссылку. Границей служит как нижний индексируемый диапазон массива, так и верхний. Часто бывает полезно умышлен-

но «заузить» адресуемый диапазон, чтобы предоставить доступ только к необходимому подмножеству значений.

10. Процесс выполнения экземпляра (виртуальной) ОС

На начальном этапе создаётся пустой контекст, в который помещается контекст главного вычислительного модуля. По традиции главным модулем должен быть модуль с именем `main`, хотя это вопрос реализации, и первым запускаемым модулем может быть просто первый модуль. Курсор текущей операции устанавливается на начало и запускается выполнение команд из текущего контекста.

10.1. Выполнение операций в контексте (текущего) вычислительного модуля

Берётся очередная операция, и в зависимости от её кода выполняются действия. Рассмотрим варианты работы для ключевых операций.

`op_call`: вызов другого вычислительного модуля. Для нового ВМ создаётся новый контекст, в который помещается отображение вызываемой функции с установкой курсора на начало операций. Кроме этого, передаются параметры в контекст вызываемого модуля. Типы и совпадение количества аргументов уже проверены на этапе загрузки кода, так что остаётся только проверить правомочность доступа и скопировать значения передаваемых аргументов. Далее осуществляются запись нового контекста во внутренний стек процессора, сдвиг курсора контекстов на единицу и переход к выполнению вновь созданного контекста. По сути, здесь идёт рекурсивный вызов, однако для нового контекста, который является подмножеством родительского.

`op_jmp`: переход к именованной метке внутри текущего вычислительного модуля. Смены контекста не происходит, следующая выполняемая операция — операция, помеченная соответствующей меткой.

`op_ret`: окончание выполнения модуля и возвращение в предыдущий контекст. Если курсор контекстов установлен на начальное положение, то происходит завершение работы данной виртуальной ОС. В противном случае уничтожается текущий контекст и сдвигается курсор на начало. Далее следует выход из рекурсии к предыдущему контексту.

`op_XXX`: какая-то операция, которую выполняет процессор. В процессе её выполнения процессор проверяет необходимые условия и ограничения. Если эта операция не последняя и не операция перехода или вызова другого ВМ, то происходит переход к выполнению следующей операции. Если эта операция последняя в ВМ, то её действие аналогично операции `op_ret`.

11. Текущее состояние и перспективы разработки

На данный момент в «композите» реализована существенная часть безопасной архитектуры. Все исходные материалы проекта доступны на странице проекта [1]. Поддерживается запуск небольших программ, успешное выполнение всех существующих на данный момент регрессионных тестов. Планируется проверить на практике работу всей системы с привлечением широкого спектра существующего прикладного обеспечения на высокоуровневых языках программирования. Для этого потребуется адаптировать существующие или разработать новые компиляторы для перевода программ с языков высокого уровня на уровень кодов архитектуры. Применение компиляторов для существующих программ, их обкатка и успешная работа в рамках предложенной архитектуры может дать толчок для реализации архитектуры в аппаратуре. В этом случае возможен отказ от использования текущей операционной системы, служащей

для запуска виртуальной машины. Кроме этого, производительность системы в целом должна существенно возрасти за счёт реализации примитивов защиты на аппаратном уровне. В остальном все положительные черты безопасной архитектуры сохраняются.

ЛИТЕРАТУРА

1. `http://skylark.tsu.ru/compozit/` — Домашняя страница экспериментальной реализации проекта Compozit. 2010.
2. `http://www.qemu.org/` — Домашняя страница проекта QEMU. 2010.
3. `http://www.vmware.com/` — Домашняя страница проекта VMWare. 2010.
4. `http://www.virtualbox.org/` — Домашняя страница проекта VirtualBox. 2010.
5. `http://www.linux.org/` — Домашняя страница проекта Linux. 2010.
6. `http://www.freebsd.org/` — Домашняя страница проекта FreeBSD. 2010.
7. `http://www.xen.org/` — Домашняя страница проекта Xen. 2010.
8. `http://www.sun.com/bigadmin/content/zones/` — Домашняя страница проекта Solaris Zones. 2010.
9. `http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/jails.html` — Документация про технологию FreeBSD jail. 2010.
10. `http://wiki.openvz.org/` — Домашняя страница проекта OpenVZ. 2010.
11. `http://www.freebsd.org/cgi/man.cgi?query=chroot&apropos=0&sektion=0&manpath=FreeBSD+8.1-RELEASE&format=html` — Руковоство по функции `chroot()` из проекта FreeBSD. 2010.