Федеральное государственное автономное образовательное учреждение высшего образования
«Московский физико-технический институт (государственный университет)»
Факультет радиотехники и кибернетики
Кафедра теоретической и прикладной информатики

Направление подготовки / специальность: 03.04.01 Прикладные математика и физика
Направленность (профиль) подготовки: Математические и информационные технологии
Форма обучения: очная

Допущен к ГИА
Руководитель учебного подразделения
_____/_____
(подпись) *расшифровка подписи*
«___» _____20__г

Исследование и реализация способов расширения номенклатуры виртуальных устройств, подключаемых к гипервизорному виртуализационному решению.

(магистерская диссертация)

Обучающийся: Фофанов Владислав Сергеевич
_____
(подпись обучающегося)
Научный руководитель: Пашков Руслан Анатольевич, к.ф.-м.н.
_____
(подпись научного руководителя)

Москва 2018

Аннотация

Цели и задачи: Данная дипломная работа ставит перед собой задачи исследования, реализации и расширения номенклатуры виртуальных устройств, подключаемых к гипервизорному виртуализационному решению. Под номенклатурой виртуальных устройств здесь и далее подразумевается программный интерфейс взаимодействия модуля устройства и гипервизорного виртуализационного решения. В части практической реализации номенклатуры акцент был сделан на устройства, использующие физический интерфейс PCI (Peripheral Component Interconnect). Конечной практической целью данной дипломной работы является реализация виртуального PCI устройства подключаемого к гипервизорному виртуализационному решению с наиболее оптимальными показателями производительности, скорости отклика, отказоустойчивости, универсальности и полноты функционала.

Полученные результаты: Результатом исследовательской части данной дипломной работы является максимально подробная информация о номенклатуре виртуальных PCI устройств: спецификация и производительность стандартных виртуальных интерфейсов (а также их аналогов, разработанных в рамках данной дипломной работы), используемые технологии увеличения производительности и пропускной способности виртуальных интерфейсов передачи данных, а также сравнение эффективности интерфейсов виртуальных устройств с аналогичными интерфейсами реальных устройств. Основным практическим результатом данной дипломной работы является реализация высокопроизводительного полнофункционального интерфейса виртуальных PCI устройств для гипервизорного виртуализационного

решения, входящего в состав Parallels Desktop, а также нескольких виртуальных PCI устройств, использующий данный интерфейс.

Рекомендации, предложенные на основании данной работы: На основании данной работы предложено расширить существующий набор интерфейсов взаимодействия виртуальных устройств за счет добавления иных эмуляционных интерфейсов (USB и т.п.), а также за счет паравиртуализационных интерфейсов, добавление которых стоит существенно облегчить по причине их разнородности и, как следствие, большого числа.

Оглавление

1. Аннотация
    1.1. Цели и задачи данной работы
    1.2. Полученные результаты
    1.3. Рекомендации, предложенные на основании данной работы
2. Содержание
3. Обозначения и сокращения
4. Введение
    4.1. Актуальность темы исследования
    4.2. Объект исследования
    4.3. Цель исследования
    4.4. Методы исследования
    4.5. Научно-техническая значимость
    4.6. Практическая значимость
5. Основное содержание
    5.1. Описание гипервизорного виртуализационного решения
    5.2. Классификация способов виртуализации
    5.3. Виртуализация памяти
        5.3.1. Теневые таблицы страниц
        5.3.2. Расширенные таблицы страниц
    5.4. Виртуализация устройств
        5.4.1. Полная эмуляция
        5.4.2. Проброс
        5.4.3. Аппаратная виртуализация
        5.4.4. Паравиртуализация
    5.5. Описание спецификации PCI устройств

5.6. Описание интерфейсов передачи команд и данных устройств
5.6.1. MMIO
5.6.2. PIO
5.6.3. DMA
5.6.4. Прерывания
5.7. Использование системы подгружаемых хостовых модулей для виртуализации устройств
5.8. Краткое введение в паравиртуализационные интерфейсы.
5.9. Реализация интерфейса MMIO для виртуализаци PCI устройств на основе системы подгружаемых модулей
5.10. Реализация интерфейса DMA для виртуализаци PCI устройств на основе системы подгружаемых модулей
5.11. Реализация прерываний для виртуализации PCI устройств на основе системы подгружаемых модулей
5.12. Сравнение производительности виртуальных интерфейсов и их аппаратно-программных аналогов
5.12.1. DMA
5.12.2. Прерывания
5.12.3. MMIO
6. Заключение
7. Список использованных источников

Обозначения и сокращения

PCI - Peripheral component interconnect.

VMM - 1)Virtual Machine Manager (Hypervizor).

2)Virtual Machine Monitor (Monitor)

Guest OS (Гостевая ОС, гость) - Операционная система, запускаемая на виртуальной машине.

Host OS (Хостовая ОС, хост) - Операционная система, установленная непосредственно на аппаратное обеспечение и в рамках которой работает гипервизор.

MMU - Memory Management Unit.

IOMMU - Input/Output Memory Management Unit.

MMIO - Memory Mapped Input/Output.

PIO - Port Input/Output.

DMA - Direct Memory Access.

VmApp - Vm Application - Хостовый процесс, отвечающий за стандартную виртуализацию устройств и связывающий виртуальные устройства с их драйверами в хостовой ОС.

Введение

Актуальность темы исследований. На сегодняшний день технологическое развитие наиболее ярко проявляет себя в сфере компьютерных технологий. Рост производительности процессоров, объема хранилищ данных, пропускной способности каналов передачи происходит в геометрической прогрессии. Параллельно с развитием самих компьютеров происходит развитие периферийных устройств различного назначения: устройства для записи/воспроизведения звука и видео, переносные носители данных… Для полноценного взаимодействия устройств с компьютером требуются эффективные интерфейсы и протоколы передачи команд/данных между ними. В случае реальных устройств необходимая производительность интерфейсов осуществляется аппаратными средствами: шинами с большей пропускной способностью и высокопроизводительными контроллерами этих шин.

Однако в этой дипломной работе будет рассмотрено решение данной проблемы для случая виртуального компьютера и виртуальных устройств. Различные технологии виртуализации повсеместно внедрены как для решения задач бизнеса, так и для бытового использования. Поэтому интерес к улучшению эффективности взаимодействия виртуальных устройств и виртуального компьютера (виртуальной машины) вполне оправдан. Основной проблемой в данном случае является невозможность прямого взаимодействия периферийных устройств с виртуальной машиной через стандартные интерфейсы. Для корректного взаимодействия требуется промежуточный программный слой, который вносит задержку в работе и потенциально уменьшает пропускную способность интерфейса передачи данных, а также может сказаться на

универсальности и полноте интерфейса. В рамках данного диплома проводятся исследования, позволяющие нивелировать или ослабить влияние вышеуказанных проблем при виртуализации устройств, которые взаимодействуют с компьютером на основе интерфейса PCI.

Объектом исследования является интерфейс передачи данных и команд между виртуальными устройствами, использующими шину PCI, и виртуальной машиной.

Целью данной дипломной работы является исследование, реализация, расширение и увеличение производительности интерфейса виртуальных PCI устройств, подключаемых к гипервизорному виртуализационному решению.

Используемые методы исследования в рамках данной дипломной работы включают в себя детальное изучение технологии гипервизорной виртуализации, технологии виртуализации памяти а также подробное изучение спецификации PCI интерфейса. Практические методы исследования заключаются в реализации теоретических концепций в рамках гипервизорного виртуализационного решения Parallels Desktop, а также реализации тестовых PCI девайсов для тестирования производительности, скорости отклика, отказоустойчивости, универсальности и полноты функционала реализованного интерфейса.

Научно-техническая значимость данной работы заключается в разработке высокоэффективных методов передачи информации между виртуальной машиной и операционной системой, на которой данная виртуальная машина развернута. Данные методы передачи информации опираются на современные протоколы упаковки, хранения и передачи данных в памяти а также на специфику виртуализации компьютеров в целом и оперативной памяти в частности с целью уменьшения накладных

расходов при передаче и достижения наибольшей пропускной способности.

Практическая значимость данной работы заключается в реализации механизма виртуализации PCI устройств для гипервизорного виртуализационного решения Parallels Desktop. Данный механизм обладает достаточно большой гибкостью и отказоустойчивостью за счет вынесения функционального модуля виртуализируемого устройства в отдельный процесс, но при этом благодаря большому объему проведенных исследований, описанных в разделе научно-технической значимости, интерфейс механизма обладает производительностью и пропускной способностью сравнимой с аппаратным интерфейсом реальных устройств. В результате с помощью данного механизма для гипервизорного виртуализационного решения Parallels Desktop можно эффективно виртуализовать практически любое высокопроизводительное PCI устройство (например современные записывающие высококачественные устройства записи видео).

Основное содержание

Описание гипервизорного виртуализационного решения. Прежде всего, вкратце рассмотрим основные принципы работы платформы для нашей исследовательской работы. Главной составляющей гипервизорного виртуализационного решения является, собственно, сам гипервизор.

Гипервизор или Virtual Machine Manager — программная схема, позволяющая одновременное, параллельное выполнение нескольких операционных систем на одном и том же хост-компьютере. Гипервизор также играет роль диспетчера ресурсов системы, таких как память, процессоры, устройства... Гипервизор обеспечивает изоляцию гостевых операционных систем друг от друга.

Гипервизоры разделяются по типу управления ресурсами хоста:

1. Гипервизор, который устанавливается непосредственно на аппаратную часть без предварительной установки операционной системы. Гипервизор такого типа, по сути, сам является «облегченной» версией хостовой операционной системы, выполняя функции распределения памяти и процессоров.

2. Гипервизор, который работает поверх установленной операционной системы в нулевом кольце (т.е. в качестве ядерного модуля). Гостевые операционные системы используют аппаратные ресурсы не напрямую, а через хостовою операционную систему (с помощью гипервизора). В некоторых случаях такие гипервизоры могут использовать алгоритмы и ресурсы операционной системы, на которой они установлены, для того, чтобы выполнять свои задачи. Например, гипервизор может использовать диспетчер процессов хостовой операционной системы в качестве диспетчера виртуальных машин,

рассматривая каждую виртуальную машину как отдельный процесс. Такой подход для встроенных гипервизоров позволяет значительно сократить расход аппаратных ресурсов на обслуживание виртуальных машин.

В рамках данной дипломной работы мы будем рассматривать гипервизоры второго типа, как наиболее распространённые. Взаимосвязь базовых компонентов гипервизорного виртуализационного решения, использующего гипервизор второго типа, изображена на рисунке 1.1.

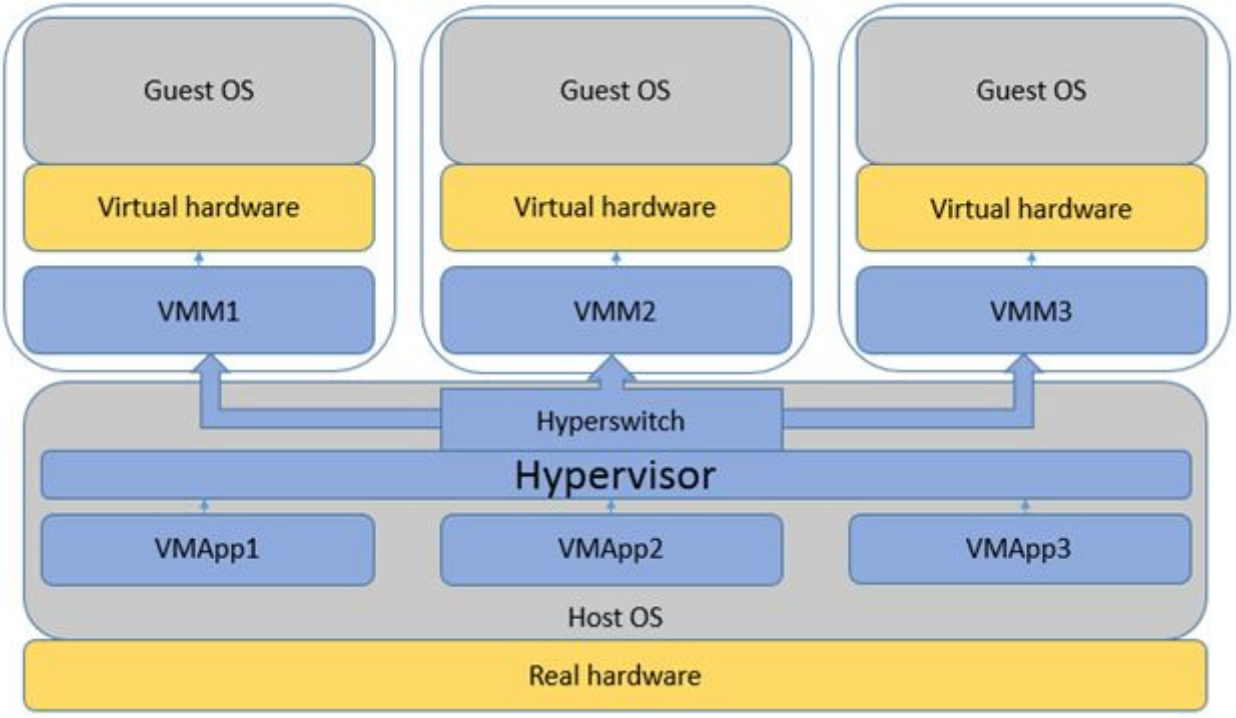

Рис. 1.1 Устройство гипервизорного виртуализационного решения

Как можно увидеть из схемы, гипервизорное виртуализационное решение состоит из следующих основных компонент:

1. Приложение уровня пользователя – процесс в хостовой операционной системе, выполняющий такие функции, как:

- Получение данных и команд от реальных устройств для последующей передачи их гостевой операционной системе.

- Работа с памятью гостевой и хостовой операционных систем, в том числе и для установления взаимодействия между ними.

- Реализация некоторых дополнительных функций гипервизорного виртуализационного решения.

2. Гипервизор – драйвер хостовой операционной системы, выполняющий две важные функции. Во-первых, гипервизор является диспетчером виртуальных машин. Он регистрирует новые виртуальные машины, отслеживает состояние уже зарегистрированных, а также отвечает за распределение системных ресурсов между виртуальными машинами. Во-вторых, гипервизор осуществляет переключение между приложением уровня пользователя и VMM (еще один компонент, используемый для виртуализации), используя механизм HyperSwitch. Данный механизм подменяет текущий контекст выполнения на контекст искусственно созданного процесса, в рамках которого исполняется VMM.

3. Монитор Виртуальных Машин или VMM – представляет из себя файл с кодом, загружаемый в искусственно созданный процесс, переключение к которому происходит с помощью механизма HyperSwitch. Основные функции VMM:

- Эмуляция тех машинных инструкций, которые не могут быть исполнены непосредственно на процессоре. Такие инструкции могут возникать вследствие различия базовых архитектур у гостевой и хостовой операционных систем, а также вследствие наличия инструкций, непосредственное исполнение которых невозможно для гостевой операционной системы.

- Эмуляция различных аппаратных событий для гостевой операционной системы, например, прерываний.

- Выполняет некоторые промежуточные действия по обработке взаимодействия между гостевой и хостовой операционными системами. Степень участия в этой обработке очень сильно зависит от способа виртуализации устройств, о чем будет подробнее рассказано в дальнейшем.

В дальнейшем, наши исследования будут широко опираться на устройство этих компонент.

Классификация способов виртуализации устройств. Виртуализацию компьютеров можно условно разделить на три направления: виртуализация процессора, виртуализация памяти и виртуализация устройств. В рамках данной дипломной работы мы будем рассматривать виртуализацию устройств и виртуализацию памяти.

Изучение виртуализации устройств является одной из основных целей данной дипломной работы. Разрабатывая новые способы виртуализации разумно опираться на уже существующие технологии, учитывать их сильные и слабые стороны. Поскольку в данном случае нас безусловно интересует практический аспект виртуализации, необходимо понимать, что разрабатывать новую технологию имеет смысл только при условии ее конкурентоспособности. Если наша технология не будет в чем-то превосходить существующие аналоги, то она окажется попросту невостребованной.

Изучение технологий виртуализации памяти необходимо для абсолютного понимания процессов, происходящих при передаче данных между гостевой ОС и виртуальным устройством. Стоит понимать, что благодаря развитой подсистеме управления памятью в большинстве современных ОС задача передачи данных не сводится к простому их перекопированию из гостевой операционной системы в хостовую. Используя поддержку в аппаратном и программном обеспечении, можно значительно сократить накладные расходы (расходы, связанные с переключением контекста исполнения кода, а также с копированием данных в промежуточные участки памяти) и увеличить пропускную способность соответствующих интерфейсов.

Далее мы более детально рассмотрим вышеобозначенные направления виртуализации, расскажем об используемых при их

реализации алгоритмах и о поддержке виртуализации со стороны операционной системы и аппаратного обеспечения.

Виртуализация памяти.

Различные технологии виртуализации памяти опираются на общую основу - использование таблиц страниц и линейного адресного пространства. Для правильного распределения физических страниц памяти и изоляции процессов друг от друга с точки зрения используемых ресурсов операционная система хранит для каждого процесса так называемые таблицы страниц. В этих таблицах содержится информация о соответствии адресов линейного адресного пространства процесса реальным физическим адресам памяти. При использовании инструкций чтения или записи в память аппаратное обеспечение “смотрит” в таблицы текущего процесса и по ним находит реальное место в физической памяти, с которым может работать инструкция.

При виртуализации памяти возникает ряд сложностей, связанный с тем, что процессор под управлением гостевой операционной системы не может напрямую выполнять необходимые действия по сопоставлению линейного и физического адресов. Для решения этой проблемы существуют два пути - программный и аппаратный.

Программный путь реализуется через так называемые *теневые таблицы страниц*. Суть этого подхода в том, что при обращении к любой ячейке памяти в гостевой операционной системе происходит переключение в контекст монитора (благодаря особой настройке гостевых таблиц страниц). В мониторе осуществляется сопоставление гостевого линейного и физического адресов но уже на основании теневых таблиц страниц, которые гостю не видны. Таким образом, мы создаем видимость полного контроля гостя над управлением памятью. Однако у данного способа есть один существенный недостаток - переключение в монитор при любом обращении в память существенно снижает производительность

эмуляции окружения гостевой ОС. Эта проблема решена не актуальна при наличии аппаратной поддержки.

При наличии аппаратной поддержки (технология носит название VT-x) к первому уровню таблиц страниц памяти в гостевой ОС добавляется второй - *расширенные таблицы страниц*. Процессор при исполнении гостевого кода осознает, что исполняет код виртуальной операционной системы. Поэтому при обращении к памяти гостя процессор осуществляет последовательно две трансляции: из линейного гостевого в физический гостевой (он же линейный хостовый) через гостевые таблицы страниц, а затем из линейного хостового в физический хостовый (то есть реальный физический) через расширенные таблицы страниц. Монитор может перехватить такие трансляции на уровне расширенной таблицы страниц, чтобы, например, эмулировать обращение к устройствам через MMIO, но при обычном обращении к памяти он этого не делает и гость работает с памятью напрямую без переключений, что решает проблему с производительностью. Именно в контексте наличия аппаратной поддержки мы будем в дальнейшем вести объяснение работы виртуализации интерфейсов.

Виртуализация устройств.

Виртуализация устройств подразделяется на четыре типа:

1. Полная эмуляция – такой тип виртуализации устройств, при котором происходит эмуляция всего стека устройства для гостевой операционной системы. Приложение уровня пользователя получает данные с устройства через какой-либо интерфейс (возможно, высокоуровневый), а затем конвертирует их в низкоуровневые запросы, которые используются непосредственно устройствами. Затем, с помощью HyperSwitch, происходит передача управления VMM, который эмулирует весь аппаратный стек данного устройства (устройство, шина, контроллер...). VMM передает запрос виртуальному устройству, и тот начинает его обрабатывать как это бы делало реальное устройство. В итоге, после обработки стеком виртуальных устройств, запрос приходит в гостевой драйвер устройства для последующей стандартной обработки. При такой модели гость полностью уверен в реальности виртуальных устройств, и, как следствие, в том, что он находиться на реальной аппаратуре.

2. Проброс – технология виртуализации устройств, при которой данные берутся непосредственно с устройства и в неизмененном виде передаются драйверу гостевой операционной системы. Для этой цели в кольцо уровня ноль хостовой операционной системы устанавливается программа-перехватчик. Она перехватывает запросы от устройств прежде чем их получит драйвер устройства хостовой операционной системы. Перехватчик посылает данные в приложение уровня пользователя, а оттуда данные передаются в VMM. VMM эмулирует событие прихода данных с устройства для гостя, таким образом передавая данные драйверу устройства гостевой операционной системы. Таким образом,

создается т.н. мост между устройством и драйвером гостя. Дополнительно, программа перехватчик посылает сигнал хостовой операционной системе об отключении устройства. Это позволяет, по сути, передать устройство под управление гостевой операционной системе.

3. Аппаратная виртуализация (Intel VT-d) – расширение техники проброса путем добавления в чипсет нового модуля – IOMMU. Данный модуль, при необходимости, перехватывает запросы устройств на DMA операции, и заменяет физическую память хостовой операционной системы на физическую память гостевой операционной системы. Таким образом, устройства могут совершать DMA операции напрямую с гостем, что значительно увеличивает производительность виртуальных устройств.

4. Паравиртуализация – техника виртуализации, при которой в качестве средства передачи данных между устройствами и гостевой операционной системой используется какой-либо нестандартный интерфейс (отличный от стандартных протоколов взаимодействия между устройствами и драйверами). Для того, чтобы реализовать такой метод, необходимо внести некоторые изменения в ядро гостевой операционной системы, чтобы она была готова к передаче данных через нестандартный интерфейс, что является одним из главных недостатков такого метода. Однако, при использовании данного метода, можно для каждого устройства наладить взаимодействие с гостевой операционной системой через свой собственный, подходящий для данного устройства, интерфейс. При этом, варьировать интерфейс можно не только в плане формата передаваемых данных, но и в плане уровня приема/передачи этих данных (более/менее высокоуровневый интерфейс).

Описание спецификации PCI устройств. В данной дипломной работе акцент сделан на виртуализацию устройств, которые используют стандарт PCI для коммуникации с компьютером. Для виртуализации устройств, использующий данный стандарт, нет необходимости переписывать драйвера в гостевой операционной системе, таким образом мы говорим именно о эмуляции, в отличие от рассматриваемой нами ранее в бакалаврском дипломе паравиртуализации через Open ToolGate. Чтобы разработать максимально эффективную систему виртуализации следует подробно изучить спецификацию PCI устройств и алгоритмы работы их интерфейсов передачи данных.

PCI (Peripheral component interconnect) шина используется для подключения периферийных устройств к материнской плате компьютера. Стандарт PCI шины включает в себя как программные так и аппаратные параметры, но поскольку в рамках данной дипломной работы мы говорим в первую очередь о виртуализации PCI устройств, то физический уровень (разъемы, разводка сигналов и т.п.) нас не интересует.

С программной точки зрения начать рассмотрение PCI стоит с инициализации устройств, подключенных к данной шине. При старте компьютера системное программное обеспечение, связанное с контроллером PCI шины, производит опрос всех подключенных к шине устройств. Компьютер узнает тип подключенных к шине устройств а также получает информацию об аппаратных ресурсах, которые требуются каждому PCI устройству. Среди аппаратных ресурсов наибольшее значение для нас будут иметь регионы физической памяти и портов, которые используются для доступа к устройству путем чтения/записи в эти регионы (технологии MMIO и PIO, которые мы рассмотрим в дальнейшем), регионы физической памяти, которые устройство использует для

чтения/записи в обход центрального процессора (технология DMA, также будет в дальнейшем рассмотрена) а также линии прерываний, по которым устройств может быстро сообщить центральному процессору о наступлении какого-либо события.

После инициализации PCI устройство начинает работать в обычном режиме, используя все вышеназванные ресурсы для обмена командами/данными/сигналами с процессором и памятью. Полная картина всех взаимодействий PCI устройства с процессором и памятью представлена на рисунке 2.

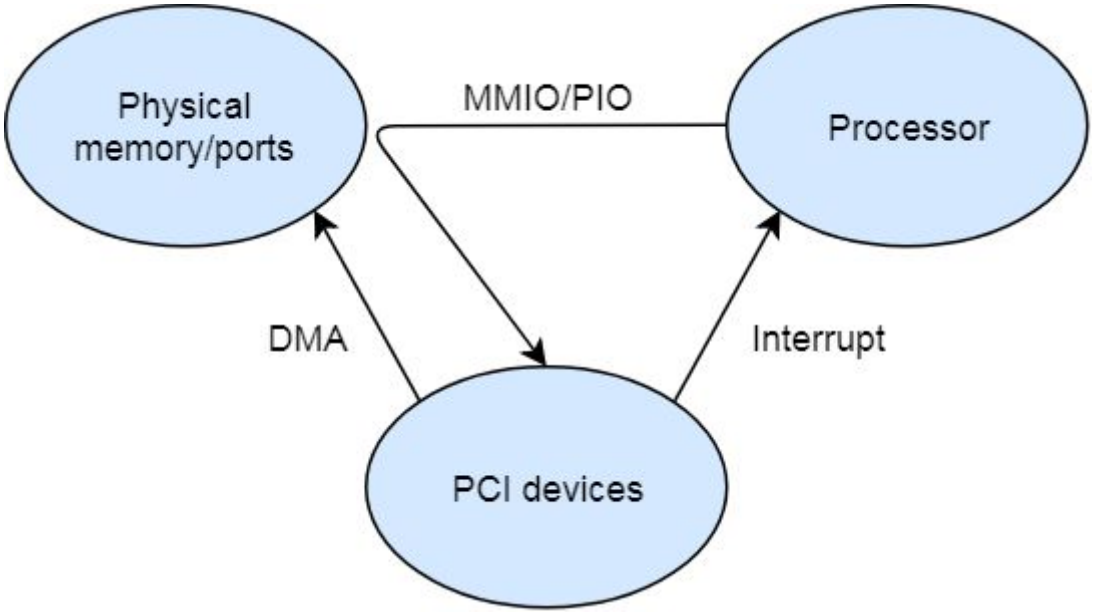

Рис. 2. Взаимодействие PCI устройств с памятью и процессором

Как мы можем увидеть из диаграммы, устройство способно взаимодействовать напрямую с процессором и памятью, но процессор может взаимодействовать с устройством только посредством интерфейса памяти. Механизм PIO также нельзя назвать прямым взаимодействием, поскольку и в этом случае используется адресное пространство - адресное пространство портов. Фактически, процессор не может напрямую отдавать произвольные команды устройствам. Устройство само задает командный

интерфейс, который процессор может использовать путем чтения/записи в соответствующую ячейку памяти MMIO или PIO.

Далее мы более подробно рассмотрим принцип работы вышеназванных интерфейсов взаимодействия.

Описание интерфейсов передачи команд и данных устройств.

1. MMIO (Memory Mapped Input/Output) – способ взаимодействия устройств с процессором через стандартный интерфейс обращения к памяти. При использовании MMIO процессор для чтения/записи в регистры устройства использует стандартные системные вызовы READ и WRITE. Процессор пишет данные по виртуальным адресам, которые поступают в MMU для конвертации в физические. В MMU адрес записи чтения конвертируется в физический. Если данный физический адрес находится в том диапазоне, который был отдан PCI устройству при инициализации, то данные направляются не на шину физической памяти, а на шину соответствующего устройства и в конечном счете попадают в память или регистры самого PCI устройства. PCI устройства могут затребовать до шести диапазонов физического адресного пространства для взаимодействия с процессором через MMIO. Несмотря на то, что размер этих областей может быть довольно большим, MMIO как правило не используется для высокоскоростного обмена данными, поскольку чтение и запись в регионы адресного пространства, принадлежащие устройству, осуществляются непосредственно процессором с помощью стандартных инструкций (например MOV). Такие инструкции оперируют сравнительно небольшими операндами, поэтому для передачи большого объема данных требуется многократно повторять эти самые инструкции, что приводит к уменьшению общей производительности системы. Обычно MMIO используется для передачи команд управления устройством и небольших объемов данных, связанных с этими командами.

2. PIO (Ports Input/Output) – способ взаимодействия с устройством через диапазоны портов ввода/вывода, отданные PCI устройству при инициализации, с использованием команд процессора IN и OUT. Поскольку данные через него передаются побайтно за одну команду, данный интерфейс имеет сравнительно небольшую скорость. Поэтому он еще менее пригоден для передачи больших объемов данных устройству.
3. DMA (Direct Memory Access) - прямой доступ к памяти из устройства. Данный метод позволяет производить обмен данными между устройством и оперативной памятью без необходимости регулярно получать инструкции от центрального процессора. При необходимости устройство может осуществить “захват” шины памяти и вести запись напрямую в те участки физической памяти, которые ему требуются. Данный способ используется для высокопроизводительного обмена данными между устройством и памятью (например в случае необходимости передать видеопоток с камеры ноутбука для запись). В более современных версиях компьютеров устройства могут вести чтение/запись данных не только по физическим, но и по логическим адресам процессора благодаря технологии IOMMU (Input/Output Memory Management Unit).
4. Прерывания (Interrupts) - интерфейс взаимодействия между устройством и центральным процессором, позволяющий устройству сигнализировать процессору о наступлении какого-либо события. При инициализации устройство получает номер линии прерывания, на которой он может выставить сигнал при необходимости. Сигнал поступает через аппаратный контроллер прерываний в процессор, который немедленно реагирует на его приход, вызывая

соответствующий обработчик прерывания. Обработчик прерывания делает необходимые для обработки поступившего сигнала действия, например начинает обрабатывать данные с камеры, записанные ею в память через DMA. Таким образом, прерывания это самый эффективный способ для устройств “попросить” процессор сделать что-либо.

Большинство из вышеописанных интерфейсов так или иначе используются при работе PCI устройств. Таким образом, виртуализация PCI устройств в первую очередь означает необходимость хорошо продуманной виртуализации вышеописанных интерфейсов с использованием ранее упомянутых технологий виртуализации. Дальнейшая часть данной дипломной работы посвящена исследованию и реализации высокопроизводительных, отказоустойчивых и универсальных интерфейсов используемых PCI устройствами.

Использование системы подгружаемых хостовых модулей для виртуализации устройств. Цель данной дипломной работы – исследование, реализация, расширение и увеличение производительности интерфейса виртуальных PCI устройств. Для реализации задуманного в рамках дипломной работы была спроектирована общая система виртуализации PCI устройств на основе подгружаемых хостовых модулей. Рассмотрим принцип работы данной системы.

В первую очередь, нам необходимо поместить информацию о загружаемых модулях (название, полный путь...) в реестр параметров виртуальной машины. Во время старта работы, виртуальная машина будет считывать данные из реестра, и на его основе составлять список подгружаемых модулей.

Далее, на основе записей из реестра, виртуальная машина загружает модули к себе (в свое адресное пространство). Однако, чтобы виртуальная машина могла взаимодействовать с кодом модуля, необходимо чтобы в модуле был реализован определенный интерфейс обратных вызовов, который виртуальная машина может использовать при необходимости. Поскольку в данной дипломной работе речь идет о виртуализации PCI устройств, то интерфейс обратных вызовов должен быть схож по функциональности со стандартным интерфейсом PCI устройств, о котором мы говорили выше. В этом случае виртуализация самого PCI устройства фактически сведётся к программной эмуляции аппаратного функционала устройства.

Основное преимущество данной модульной системы в том, что при прописанном заранее формате интерфейса обратных вызовов и доступе к реестру виртуальной машины, ее может использовать пользователь без

доступа к исходному коду виртуальной машины. Это позволит разработчикам виртуализировать произвольные устройства, используя модули в качестве эмулируемого устройства или прослойки между реальным устройством и гостевой операционной системой.

Также стоит отметить высокий уровень отказоустойчивости данной системы. Фактически мы получаем дополнительный уровень изоляции устройства, который в случае критических ошибок в модуле устройства позволяет сэмулировать более мягкий механизм его отключения, позволяя избегать неполадок в работе виртуальной операционной системы. Даже если по каким-то причинам процесс модуля аварийно завершит свою работу, та часть VmApp, которая отвечает за связь с модулями и их менеджментом, заметит отключение одного из виртуальных устройств и соответствующим образом сэмулирует корректное завершение работы этого устройства в госте. Таким образом , проблемы в одном из модулей никогда не будут приводить к падению гостевой операционной системы.

Наконец, стоит упомянуть о существовании возможности расширить PCI интерфейс виртуальных устройств за счет паравиртуализационных интерфейсов обмена командами и данными (исследование таких интерфейсов производилось в бакалаврской работе). При использовании хотя бы одного такого интерфейса виртуализация устройства несколько усложнится, поскольку возникнет необходимость переписать драйвера устройства в гостевой операционной системе с учетом поддержки паравиртуализации. Однако эффект от использования данного интерфейса (с учетом возможности его подстройки под конкретный класс устройств) вполне может окупить производственные затраты.

Фактически, задача реализации интерфейса между виртуальными PCI устройствами и виртуальной машиной сводится к программной

реализации интерфейса, предоставляемого подгружаемым модулям, которые содержат эмулируемые PCI устройства. Исходя из этого положения мы и будем разрабатывать конкретные протоколы эмуляции ранее рассмотренных интерфейсов для взаимодействия PCI устройств с центральным процессором и памятью.

Краткое введение в паравиртуализационные интерфейсы. Для более детального понимания преимуществ и недостатков эмуляции PCI устройств по сравнению с виртуализацией, вкратце рассмотрим исследуемую в бакалаврской работе тему виртуализации устройств при помощи паравиртуализацтионного интерфейса Open ToolsGate.

Open ToolsGate - реализованный в рамках Parallels Desktop паравиртуализационный интерфейс передачи данных. Концептуальная схема расширенного варианта данного интерфейса для передачи данных между гостевой и хостовой операционными системами представлен на рисунке 2.1.

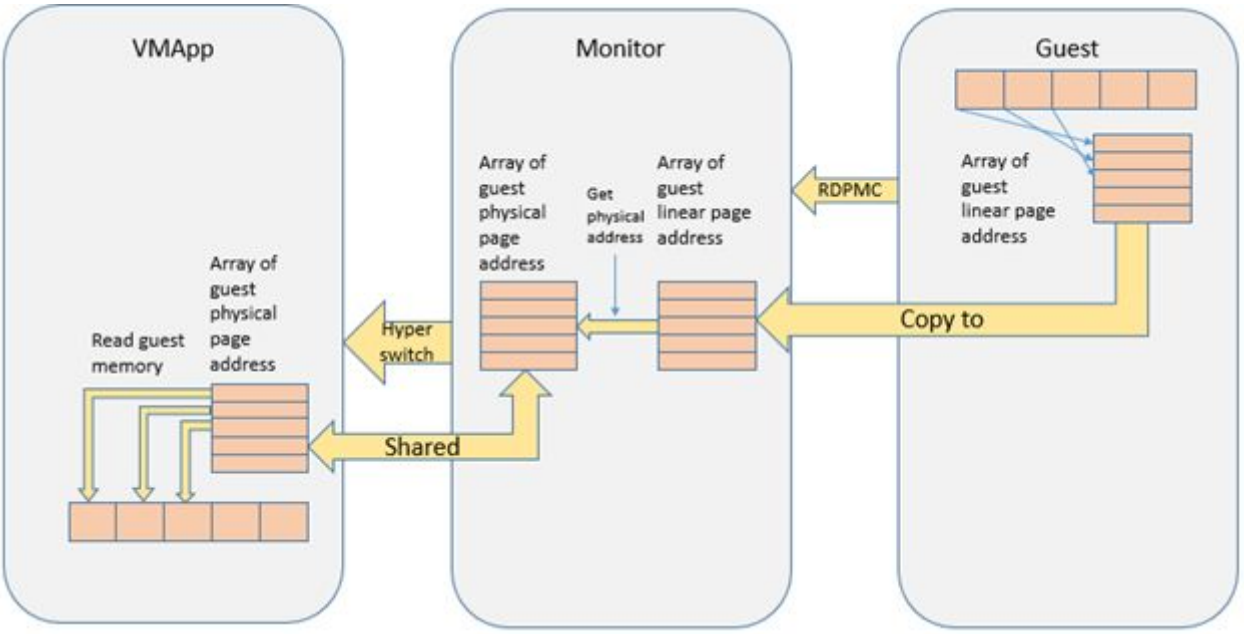

Рис 2.1. Схема расширенного варианта интерфейса Open ToolsGate.

В расширенном варианте интерфейс, предоставляемый пользователю, аналогичен интерфейсу базового варианта (тот же API, также поставляется в виде библиотека + заголовочный файл). Различие заключается в способе передачи данных.

Когда пользователь вызывает функцию расширенного интерфейса для передачи данных, функция не передает буфер по 4-рех килобайтным сегментам (как это было в базовом механизме). Вместо этого, в 4-рех килобайтный буфер, используемый для передаваемых данных при вызове RDPMC, записываются адреса виртуальных страниц, на которых лежит гостевой пользовательский буфер. На каждую запись выделяется 4 байта из 4096. Адрес записывается не напрямую, а с 12-битным смещением (т.к. это адрес страницы, то эти биты нулевые). Это позволяет использовать 44-битный диапазон адресов, т.е. 16 Терабайт памяти.

После вызова RDPMC и передачи управления в VMM, вызывается обработчик OTG запросов на стороне VMM. Внутри него мы используем интерфейс виртуальной машины для получения физического адреса тех страниц, линейные адреса которых ранее были записаны в буфер обмена. В итоге, проделав данную операцию для всех адресов из списка, мы получаем аналогичный буфер, только заполненный уже гостевыми физическими, а не виртуальными, адресами. Дополнительно, в рамках получения физического адреса страницы, делается проверка на присутствие страницы в физической памяти. Если страница отсутствует, то мы приостанавливаем выполнение кода обработчика, а затем эмулируем PAGE_FAULT для данной страницы в гостевой операционной системе. Гость обрабатывает данный PAGE_FAULT, а затем перезапускает инструкцию RDPMC, чтобы мы могли корректно получить гостевой физический адрес страницы. После того, как адреса всех страниц из списка конвертированы в гостевые физические, мы перекопируем буфер с адресами в буфер, разделяемый VMM и приложением уровня пользователя, и передаем второму управление.

В приложении уровня пользователя, опять же, используется интерфейс виртуальной машины для считывания содержимого гостевых физических страниц (или для записи в них). В итоге, мы считываем содержимое всех физических страниц, содержащихся в списке, в специальный хостовый буфер, который передаем обработчику на стороне хоста. Возвращаемые данные передаются таким же способом (через запись по физическим адресам из списка).

В результате, данный метод устранил все недостатки базового варианта интерфейса для обмена данными. При использовании расширенного метода, нам не нужно копировать все данные во время обработки в VMM (мы копируем только 4-рех килобайтный буфер со списком адресов). Также, поскольку адреса страниц буфера передаются на хостовую операционную систему за один вызов RDPMC (если размер буфера данных не превышает 4 Мегабайта), то и VMexit, и HyperSwitch происходит всего один раз за весь цикл обмена данными (при использовании базового метода, при передаче 4 Мегабайт данных, число переключений контекста равно 1024). Благодаря этому, скорость передачи данных расширенного метода должна значительно превосходить скорость базового метода.

Стоит также заметить, что в случае данных, размером более 4 Мегабайт, требуется надстройка над расширенным механизмом передачи данных, которая будет разбивать буфер на 4-рех Мегабайтные сегменты и передавать их по отдельности. Данная надстройка, по своей концепции, практически идентична концепции базового интерфейса передачи данных.

Далее мы будем рассматривать альтернативный вариант виртуализации устройств - эмуляцию.

Реализация интерфейса MMIO для виртуализаци PCI устройств на основе системы подгружаемых модулей. На рисунке 3 представлена концептуальная схема виртуализации интерфейса MMIO, используемая для модулей виртуальных PCI устройств.

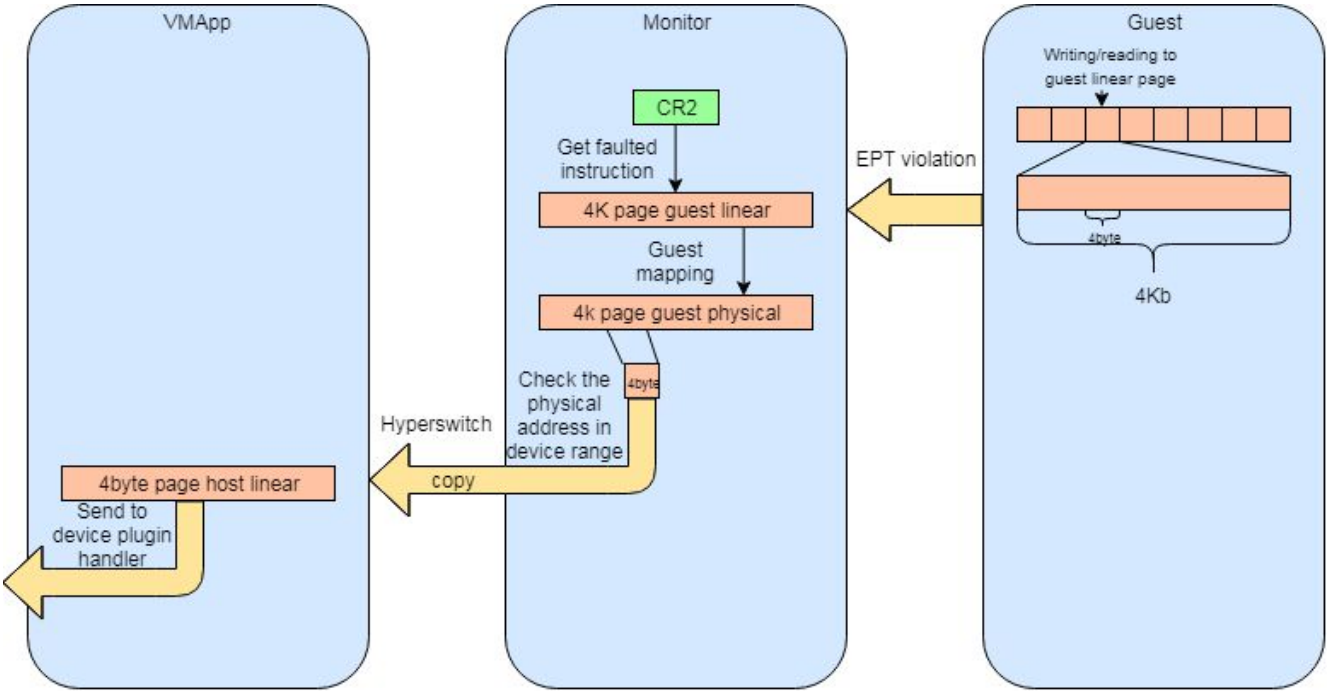

рис. 3. Схема виртуализации интерфейса MMIO.

Принцип работы данной схемы виртуализации основывается на использовании возможностей технологии аппаратной виртуализации Intel EPT. Гостевой линейный адрес памяти, через который мы хотим доступиться в устройство, проходит два уровня трансляции. Первый уровень - трансляция из гостевого линейного адреса в гостевой физический - стандартный и проводится непосредственно гостевой операционной системой. Второй уровень - трансляция из гостевого физического адреса в хостовый физический обеспечивается аппаратной поддержкой в процессоре и позволяет нам перехватить обработку доступа к данным и направить ее в монитор.

В мониторе мы, получив гостевой физический адрес, проверяем диапазоны гостевых физических адресов, отданных эмулируемым устройствам. Когда мы понимаем, что полученный физический адрес входит в диапазон одного из эмулируемых устройств, мы с помощью стандартного интерфейса обмены сообщениями между монитором и VmApp направляем обрабатываемый запрос в VmApp. Внутри себя VmApp определяет, к какому эмулируемому устройству относится данный запрос и передает обработку ему.

Как отчетливо видно из вышеописанного алгоритма, обмен данным через MMIO осуществляется малым порциями (4 или 8 байт в зависимости от инструкции, которую мы используем в госте для чтения или записи). Притом при каждом цикле передачи данные при эмуляции проходят довольно большой путь, во время которого монитор и VmApp совершают множество ресурсозатратных действий (VmExit при EPT Violation, проверка всех диапазонов гостевой физической памяти устройств, HyperSwitch при передаче данных из монитора в VmApp). Поэтому данный интерфейс используется лишь для управления PCI устройствами путем посылки команд от процессора. Как следствие, требования к высокопроизводительной эмуляции в данном случае отсутствуют.

Механизма передачи данных через порты ввода/вывода - PIO - фактически идентичен MMIO как в плане инициализации, так и в плане алгоритма работы с ним. поэтому уместно будет рассказать о нем в этом разделе. Однако по причине его более раннего появления он долгое время использовался для передачи данных от процессора к устройству и наоборот. Когда процессор хотел обменяться данными с устройством, он циклически производил чтение или запись соответствующего порта с помощью инструкций IN и OUT. Данные инструкции также оперировали

данными размером 4 байта, поэтому процедура чтения или записи устройства серьезно замедляла работу компьютера. В дальнейшем, при появлении DMA, интерфейс PIO перестал использоваться для передачи данных и, аналогично MMIO, начал использоваться для передачи команд PCI устройствам.

Реализация интерфейса DMA для виртуализаци PCI устройств на основе системы подгружаемых модулей. На рисунке 4 представлена концептуальная схема виртуализации интерфейса DMA, используемая для модулей виртуальных PCI устройств.

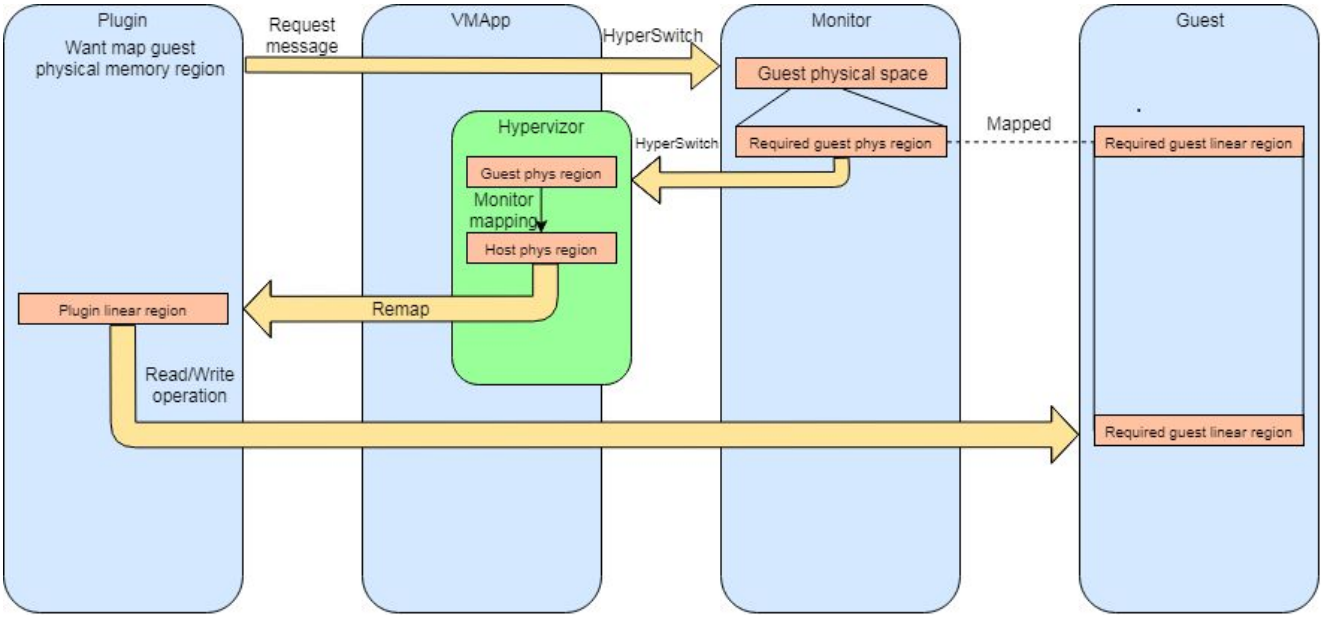

Рис. 4. Схема виртуализации DMA интерфейса.

Из схемы видно, что для того, чтобы иметь возможность доступаться до физической памяти гостя, виртуальное устройство должно вызвать специальную функцию, которая сопоставит необходимый участок физической памяти гостя и аналогичный по размеру непрерывный участок линейного адресного пространства плагина. Данная функция совершает последовательное переключение из плагина в VmApp и из VmApp в монитор. В мониторе производится поиск соответствующего региона гостевой физической памяти, проверяется валидность выбранного диапазона. Если такой регион существует, и он (с точки зрения гостевой операционной системы) принадлежит оперативной памяти, то управление передается в гипервизор. Данное переключение является ключевым, поскольку необходимый для сопоставления физических гостевых и

линейных хостовых диапазонов функционал присутствует только в контексте работы ядра хостовой операционной системы.

После переключения в гипервизор мы выясняем, каким реальным физическим страницам соответствует переданный гостевой физический диапазон. Данное действие во многом зависит от способа виртуализации памяти, однако всестороннее рассмотрение этого вопроса выходит за рамки данной дипломной работы. Мы рассмотрим лишь частный случай, в рамках которого производилось тестирование вышеописанных технологий виртуализации - nested paging. В нем концепция виртуальной памяти гостя реализуется через двойное отображение - сначала гостевое линейное адресное пространство через стандартную таблицу страниц отображается в гостевое физическое адресное пространство, а затем гостевое физическое адресное пространство использует дополнительное отображение для сопоставления себя с реальными физическими адресами. При такой технологии гостевые физические адреса фактически рассматриваются как хостовые линейные, а гостевое физическое адресное пространство является частью линейного пространства монитора. Поэтому, зная гостевой физический адрес мы можем, используя мониторное отображение, получить соответствующий ему реальный физический адрес.

Получив в гипервизоре реальный физический адрес, мы настраиваем отображение в процессе плагина таким образом, чтобы сопоставить полученные физические адреса с линейными адресами данного процесса, а после возвращаем управление плагину через VmApp. Теперь плагин имеет возможность через полученные линейные адреса в своем адресном пространстве писать (или читать) в физические адреса, соответствующие запрошенным гостевым физическим адресам, то есть фактически

доступаться к физической памяти гостя в обход тех процессоров, которые исполняют гостевой код, или по-другому совершать DMA операции.

Данный интерфейс используется в первую очередь для передачи больших объемов данных в единицу времени, поскольку при настроенных DMA регионах устройство может напрямую обмениваться с ними данными, не отвлекая при этом центральный процессор. Поэтому для эффективной виртуализации данного интерфейс необходимо как можно сильнее уменьшить накладные расходу при чтении/записи.

Как видно из описания виртуализационного решения, дополнительные накладные расходы (переключения контекстов, копирование данных) в нашем случае появляется только при настройке DMA регионов. При работе с уже настроенными регионами накладные расходу отсутствуют как таковые - чтение и запись ведется из плагина устройства напрямую, без перекопирований и переключений контекстов. Таким образом, можно с уверенностью говорить об успешности выбранной технологии виртуализации интерфейса DMA.

Реализация прерываний для виртуализации PCI устройств на основе системы подгружаемых модулей. На рисунке 5 представлена концептуальная схема виртуализации прерываний, используемая для модулей виртуальных PCI устройств.

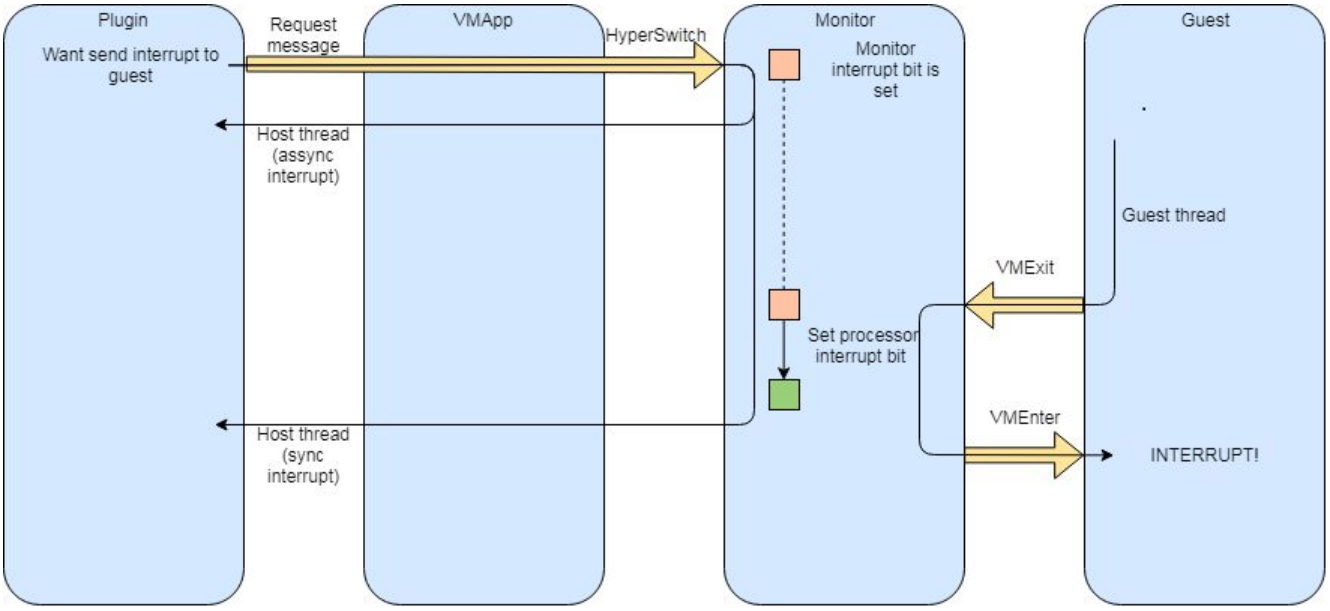

Рис. 5. Схема виртуализации прерываний.

Интерфейс прерываний служит лишь одной цели - отправить сигнал процессору от устройства о наступлении какого-либо события, которое процессор должен обработать. Поэтому эмуляция данного интерфейса фактически сводится к выставлению бита прерываний на гостевом процессоре.

Основная проблема, возникающая в данном случае, заключается в трудности выставления бита прерывания гостевого процессора “со стороны”, т. е. не с самого гостевого процессора. Решить эту проблему можно двумя путями.

Первый путь реализуем не всегда - он требует аппаратной поддержки в виде межпроцессорных прерываний. При наличии такой аппаратной поддержки хостовый процессор при переключении в монитор

может просто послать межпроцессорное прерывание гостевому процессору. Получив его, гостевой процессор переключится в монитор, где межпроцессорное прерывание может быть спокойно заменено прерыванием от устройства.

Однако, как уже было сказано, данный способ требует аппаратной поддержки, которая есть не во всех процессорах.

При отсутствие аппаратной поддержки для виртуализации прерывании может быть использован второй способ (изображенный на рисунке 5). Его суть заключается в том, что после переключения в монитор хостовый тред не будет немедленно извещать гостевой процессор о посланном от устройства прерывании, а лишь выставит мониторный бит соответствующего прерывания. Когда гостевой процессор переключится в монитор (причина нам в данном случае не важна), он обратит внимание на этот бит и самостоятельно проставит себе необходимые биты для вызова прерывания от устройства. В данном случае гостевой процессор может получить прерывание лишь через некоторое время после его посылки, поэтому имеет смысл говорить о двух режимах работы виртуализированного интерфейса прерываний: синхронный и асинхронный. В первом случае хостовый тред после выставления мониторного бита прерываний не уходит сразу из монитора, а ждет, пока гостевой процессор подхватит данное прерывание. Соответственно, во втором случае, хостовый тред сразу после выставления мониторного бита переключается в плагин. Необходимость в обоих этих вариантах имеет место быть, поэтому оба они реализованы на практике.

По итогу виртуализацию данного интерфейса взаимодействия виртуального устройства и виртуальной машины нельзя назвать полностью успешной, поскольку для высокопроизводительной работы

устройства крайне важна быстрая реакция центрального процессора на выставленные прерывания. В нашем случае при выставлении прерываний мы совершаем большое число ресурсозатратных переключений контекстов, что приводит к увеличению накладных расходов. Но еще хуже то, что при отсутствии аппаратной поддержки межпроцессорных прерываний мы вынуждены ждать, пока гостевой процессор не исполнит “опасную” инструкцию, которая переключит его в монитор. Это вносит чрезвычайно большие задержки в процесс передачи прерывания.

Сравнение производительности виртуальных интерфейсов и их аппаратно-программных аналогов. Описанные выше способы виртуализации интерфейсов PCI устройств являются наиболее эффективными из тех, что были рассмотрены в рамках исследований, проводимых для данной дипломной работы. Последовательное устранение технологических недоработок на протяжении всей исследовательской деятельности привели к созданию оптимальных алгоритмов виртуализации (с точки зрения производительности, скорости отклика, отказоустойчивости, универсальности и полноты функционала).

Тем не менее, говорить об практической эффективности модели виртуализации имеет смысл только в сравнении с реально функционирующими аналогами виртулизируемого функционала. В первую очередь имеется ввиду стандартный механизм взаимодействия PCI устройств (не виртуальных) и компьютера. Также, поскольку практическая реализация виртуализации интерфейса PCI устройств разработана для гиервизорного виртуализационного решения Parallels Desktop, которое представляет собой полностью функционирующую виртуальную машину, то имеет смысл сравнить производительность с той виртуализацией интерфейса, которая уже реализована в рамках данного гипервизорного виртуализационного решения.

DMA:

Для тестирования скорости передачи данных через интерфейс DMA для каждой из вышеописанных моделей требуется применить различный подход. Для тестирования эмуляции, описанной в данной дипломной работе, был реализован специальный модуль тестового устройства, которое передает через DMA блоки данных различного размера и взаиморасположения. Для тестирования эмуляции DMA в составе Parallels Desktop создание тестового устройства не требуется - оно уже реализовано в рамках тестовой сборки. Наконец, для тестирования скорости реального DMA мы используем <выбрать технологию тестирования>. Для полноты картины мы будем тестировать скорость чтения/записи через DMA в нескольких различных режимах. Результаты тестов отражены в таблице 1.

	1 Byte consistent	4 Byte consistent	4 KByte consistent	1 Byte random	4 Byte random	4 KByte random
Our DMA virtualization	7,55 Gb/s	8,02 Gb/s	8,32 Gb/s	51 Mb/s	196 Mb/s	8,23 Gb/s
Parallels Desktop DMA virtualization	7,43 Gb/s	7,21 Gb/s	8,77 Gb/s	101 Mb/s	245 Mb/s	8,42 Gb/s

Табл. 1. Скорость работы DMA интерфейса.

Из результатов видно, что между двумя различными реализациями виртуализации DMA интерфейса практически нет разницы в производительности. Это вполне объясняется тем, что в обоих случаях после инициализации региона DMA запись в него ведется напрямую, без перекопирований и переключений контекста.

Прерывания:

В качестве единицы измерения эффективности виртуализации данного интерфейса возьмем время от отправки прерывания до получения этого прерывания гостевой операционной системой. Для подсчета этого времени в нашем варианте виртуализации мы используем формулу:

$$t_{\text{прер}} = t_{\text{синх}} - \frac{t_{\text{асинх}}}{2}$$

где $t_{\text{синх}}$ - время синхронного запроса на прерывание, а $t_{\text{асинх}}$ - время асинхронного запроса. Из рисунка 5 легко увидеть, что данная формула дает нам искомую величину. Для тестирования эмуляции Parallels Desktop мы опять же будем использовать тестовую сборку, причем на двух разных компьютерах - с поддержкой межпроцессорных прерываний и без нее. Результаты тестирования отражены в таблице 2. Для наглядности вместо времени запроса взята обратная величина - максимальная частота запросов.

	t_{sync}	t_{async}	f_{int}
Our interrupt virtualization	11,3 мкс	1,2 мкс	93000
Parallels Desktop interrupt virtualization (without IPI)	-	-	176000
Parallels Desktop interrupt virtualization (with IPI)	-	-	543000

Табл. 2. Скорость работы прерываний.

Как видно из результатов, наша технология виртуализации хоть и проигрывает аналогу из Parallels Desktop, но все же сопоставима с ней при отсутствии поддержки межпроцессорных прерываний. С наличием межпроцессорных прерываний при посылке прерывания время

нахождения в мониторе сокращено до минимума, поэтому общая скорость отправки прерываний увеличена на порядок по сравнению с нашей виртуализацией.

MMIO

Поскольку MMIO является интерфейсом передачи команд в большей степени, чем интерфейсом передачи данных, то и тестировать его эффективность более уместно с точки зрения скорости передачи команд, чем с точки зрения пропускной способности. Для данного типа тестов мы опять воспользуемся нашим тестовым плагином и тестовой сборкой Parallels Desktop. Однако, в отличие от предыдущих тестов, измерения времени мы будем проводить не в тестовых плагинах, а в гостевой операционной системе, поскольку при использовании данного интерфейса инициатором взаимодействия всегда выступает операционная система, а не устройство. Результаты тестирования отражены в таблице 3 (В качестве единицы измерений снова выбрана максимальная частота вызовов).

	$f_{request}$
Our MMIO virtualization	32114
Parallels Desktop MMIO virtualization	54096

Табл. 3. Скорость работы MMIO.

Как видно из результатов тестирования, время отправки данных через интерфейс MMIO в случае нашей виртуализации несколько больше, чем в случае виртуализации Parallels Desktop. Это объясняется тем, что в нашей реализации виртуальные устройства вынесены в отдельный процесс, поэтому при MMIO запросе системе требуется сделать дополнительное переключение контекста из процесса плагина в VmApp.

Поскольку в нашей модели виртуализации интерфейсы MMIO и PIO эмулируются одинаково (по причине их практически полной идентичности),

то измерять производительность PIO отдельно не имеет смысла - она идентична производительности MMIO.

Как мы можем увидеть, показатели производительности виртуализированных интерфейсов у нашей модели в целом уступают аналогичным показателям модели Parallels Desktop. Это объясняется концептуальным решением вынести эмуляцию PCI устройств в отдельные процессы-плагины с целью увеличить отказоустойчивость системы и упростить разработку новых виртуальных PCI устройств.

Заключение

Целью данной дипломной работы является исследование, реализация, расширение и увеличение производительности интерфейса виртуальных PCI устройств, подключаемых к гипервизорному виртуализационному решению. В каждом из четырех вышеобозначенных направлений удалось достичь значительных результатов.

В рамках исследования виртуальных интерфейсов были рассмотрены существующие аналоги виртуализации устройств, оценены достоинства и недостатки используемых ими интерфейсов передачи команд и данных. Также проводилось исследование со стороны виртуализируемой операционной системы, к которой подключались вышеназванные устройства. Были рассмотрены различные варианты предоставляемого подключаемым устройствам функционала, учтены их сильные и слабые стороны. На основании результатов исследований с учетом всех параметров была разработана наиболее оптимальная система виртуализации сторонних устройств через подключаемые модули, опирающаяся на интерфейс PCI.

Результатом программной реализации данной модели является эффективное сочетание менеджера виртуализируемых устройств и интерфейса обмена данными и командами между устройством и гостевой ОС. По сравнению с аналогами, данная модель имеет ряд преимуществ:

1) Модульность - хостовая часть виртуализации устройства требует написание лишь одного модуля-плагина. Предоставляемое модулю окружение позволяет создавать цельные эмуляции устройств, что значительно упрощает разработку таких виртуальных устройств.
2) Изолированность - модуль виртуального устройства представляет собой отдельный процесс в рамках хостовой операционной системы.

При возникновении ошибки во время работы, такой модуль позволяет безопасно завершить работу устройства, тем самым предотвращая падение гостевой операционной системы. Даже если ошибка в работе модуля приведет к завершению соответствующего хостового процесса, VmApp сэмулирует “правильное” отключение устройства.

3) Гибкость - реализованный вариант интерфейса предоставляет вариативность в способе виртуализации устройства. Если разработчик хочет просто эмулировать свое PCI устройство, он может использовать стандартный PCI интерфейс и устройство будет представлено гостевой ОС как PCI устройство, что позволит использовать стандартные драйвера данного устройства. Если же разработчик хочет использовать один из паравиртуализационных интерфейсов, он может соответствующим образом инициализировать свой плагин, что позволит ему использовать дополнительные каналы связи с гостевой ОС, но в этом случае ему придется написать альтернативный драйвер своего устройства, поддерживающий использование выбранных дополнительных каналов.

Расширение интерфейса виртуальных PCI устройств было произведено за счет добавления альтернативных паравиртуализационных каналов передачи команд и данных между устройством и гостевой ОС. Данные каналы имеют специфические алгоритмы передачи данных и могут быть использованы для каких-либо индивидуальных задач конкретных устройств. Интерфейс виртуальных устройств в целом устроен так, что добавление новых паравиртуализационных каналов не вызовет больших сложностей. Таким образом, при необходимости каждой группе

устройств могут быть сопоставлены подходящие паравиртуализационные каналы.

Увеличение производительности интерфейсов было достигнуто за счет применения новых алгоритмов упаковки и передачи данных, которые учитывают виртуализационные особенности гипервизорного решения: накладные расходы при переключении контекстов, взаимное соответствие гостевой и хостовой памяти, используемые гостевой ОС модели управления памятью…

В результате полученная программная разработка представляет из себя не только грамотную реализацию предложенной модели виртуализации устройств, но и вполне конкурентоспособный продукт, превосходящий по многим параметрам существующие решения для виртуализации устройств. В перспективе данный продукт будет включен в состав гипервизорного виртуализационного решения Parallels Desktop.

Список использованных источников

Edouard Bugnion, Jason Nieh, Dan Tsafrir Hardware and Software Support for Virtualization (Synthesis Lectures on Computer Architecture). Morgan & Claypool publishers, 2017. - 183 с.

Intel® 64 and IA-32 architectures software developer's manual. 2016. - 4844 с.

John Hoopes Virtualization for Security: Including Sandboxing, Disaster Recovery, High Availability, Forensic Analysis, and Honeypotting. - 1 изд. Syngress, 2008. - 384 с.

Branden R. Williams, Anton Chuvakin PCI Compliance, Third Edition: Understand and Implement Effective PCI Data Security Standard Compliance. - 3 изд. Syngress, 2012. - 360 с.

Dave Shackleford Virtualization Security: Protecting Virtualized Environments. - 1 изд. Sybex, 2012. - 360 с.

Tim Cerling, Jeffrey L. Buller Mastering Microsoft Virtualization. - 1 изд. Sybex, 2009. - 576 с.

Vaughn Stewart, Michael Slisinger Virtualization Changes Everything: Storage Strategies for VMware vSphere & Cloud Computing. CreateSpace Independent Publishing Platform, 2012. - 260 с.

Barry B. Brey The Intel Microprocessors 8086/8088, 80186/80188, 80286, 80386, 80486, Pentium, and Pentium Pro Processor Architecture, Programming, and Inter- facing. - 6 изд. Prentice Hall, 2002. - 1024 с.

Vasile Lungu Assembly Language Programming for Intel Processors Family. Teora USA, LLC, 2005. - 578 с.

Benjamin Ray Seyfarth Introduction to 64 Bit Intel Assembly Language Programming for Linux: Second Edition. - 2 изд. CreateSpace Independent Publishing Platform, 2012. - 308 с.