был снова переработан и реализован в виде **VM/370** для серии машин System/370, выпущенной в 1972 году. Линейка машин System/370 в 1990-х годах была заменена компанией IBM линейкой System/390. В основном изменилось только название, поскольку базовая архитектура из соображений обратной совместимости осталась прежней. Разумеется, аппаратные технологии стали совершеннее и более новые машины были больше и быстрее старых, но что касается виртуализации, ничего не изменилось. В 2000 году IBM выпустила z-серии, поддерживающие 64-разрядное виртуальное адресное пространство при сохранении обратной совместимости с System/360. Все эти системы поддерживали виртуализацию на десятилетия раньше того момента, когда она приобрела популярность на машинах семейства x86.

В 1974 году двое ученых из Калифорнийского университета (Лос-Анджелес), работающих в компьютерной сфере, Геральд Попек (Gerald Popek) и Роберт Голдберг (Robert Goldberg), опубликовали основополагающую статью («Formal Requirements for Virtualizable Third Generation Architectures»), в которой дали точный перечень тех условий, которым должна отвечать компьютерная архитектура, чтобы иметь возможность эффективно поддерживать виртуализацию (Popek and Goldberg, 1974). Написать главу о виртуализации без ссылки на их работу и терминологию просто невозможно. Примечательно, что широко известная архитектура x86, которая также берет начало в 1970-х годах, десятилетиями не отвечала этим требованиям. Но это было еще не все. Практически каждая архитектура со времен универсальных машин (мейнфреймов) также проваливала тест. 1970-е годы были весьма продуктивными, они увидели рождение UNIX, Ethernet, Cray-1, Microsoft и Apple, — следовательно, несмотря на то что могли бы сказать ваши родители, в 1970-е на планете гремел не только стиль диско!

Фактически настоящая революция **Disco** грянула в 1990-е годы, когда исследователи из Стэндфордского университета разработали новый гипервизор с таким именем и стали основателями **VMware**, гиганта виртуализации, предлагающего гипервизоры типа 1 и типа 2 и теперь гребущего миллиарды долларов дохода (Bugnion et al., 1997; Bugnion et al., 2012). Кстати, разница между гипервизорами типа 1 и типа 2 также пришла из 1970-х (Goldberg, 1972). VMware представила свое первое решение по виртуализации для x86 в 1999 году. Затем последовали и другие продукты: **Xen**, **KVM**, **VirtualBox**, **Hyper-V**, **Parallels** и многие другие. Похоже, время для виртуализации как раз подоспело, даже при том, что теорию застолбили еще в 1974 году, а IBM десятилетиями продавала компьютеры, поддерживающие и активно использующие виртуализацию. В 1999 году виртуализация приобрела широкую популярность, но несмотря на внезапно возросший к ней массовый интерес, новинкой она не была.

7.2. Требования, применяемые к виртуализации

Важно понимать, что виртуальные машины работают так же, как и реальные. В частности, у них должна быть возможность начальной загрузки, как на реальных машинах, и установки на них произвольных операционных систем, точно так же, как это может быть сделано на реальном оборудовании. Предоставление этой иллюзии с обеспечением достаточной эффективности является задачей гипервизора. Несомненно, гипервизоры должны хорошо проявлять себя по трем направлениям:

1. **Безопасность** — у гипервизора должно быть полное управление виртуализированными ресурсами.

2. **Эквивалентность** — поведение программы на виртуальной машине должно быть идентичным поведению этой же программы, запущенной на реальном оборудовании.
3. **Эффективность** — основная часть кода в виртуальной машине должна выполняться без вмешательства гипервизора.

Несомненно, безопасный способ выполнения инструкций заключается в поочередном рассмотрении каждой инструкции в **интерпретаторе** (например, в Bochs) и в выполнении именно того, что нужно для данной инструкции. Некоторые инструкции могут быть выполнены напрямую, но их не много. Например, интерпретатор может быть способен выполнить инструкцию INC (инкремент) просто как есть, но инструкции, которые небезопасно выполнять напрямую, должны быть эмулированы интерпретатором. Например, нельзя разрешать гостевой операционной системе блокировать прерывания для всей машины или модифицировать отображения страниц в таблицах. Нужно применить прием, заставляющий операционную систему, посаженную поверх гипервизора, полагать, что она заблокировала прерывания или изменила отображение страниц на машине. Позже будет показано, как это делается. А сейчас хочется лишь сказать, что интерпретатор может быть безопасным и при тщательной реализации, возможно, даже высококачественным, но производительность его может оказаться не на высоте. Далее будет показано, что для того, чтобы соответствовать также критериям производительности, мониторы виртуальных машин (VMM) стараются выполнить основную часть кода непосредственным образом.

Теперь обратимся к точности. Виртуализация долгое время была проблемой для архитектуры x86 из-за дефектов в архитектуре Intel 386, которые упорно в течение 20 лет переносились на новые центральные процессоры во имя обратной совместимости. В двух словах, каждый центральный процессор с режимом ядра и пользовательским режимом имеет набор инструкций, ведущих себя по-разному в зависимости от того, в каком режиме они выполняются, в режиме ядра или в пользовательском режиме. В их число входят инструкции, осуществляющие ввод-вывод, изменяющие настройки блока управления памятью (MMU) и т. д. Попек и Голдберг назвали их **служебными инструкциями** (sensitive instructions), или инструкциями, чувствительными к виртуализации. Есть также набор инструкций, которые при выполнении в пользовательском режиме вызывают системные прерывания. Попек и Голдберг назвали их **привилегированными инструкциями** (privileged instructions). В статье этих специалистов впервые утверждалось, что машина может быть подвергнута виртуализации, только если служебные инструкции являются поднабором привилегированных инструкций. Проще говоря, при попытке сделать в пользовательском режиме то, что вы не должны делать в этом режиме, оборудование должно вызвать системное прерывание. В отличие от IBM/370, обладающей эти свойством, у Intel 386 его нет. При выполнении в пользовательском режиме будут проигнорированы или выполнены по-другому многие служебные инструкции 386-е машины. Например, инструкция POPF заменяет регистр флагов, который изменяет бит, благодаря которому блокируются и разблокируются прерывания. В пользовательском режиме этот бит просто не изменяется. Вследствие этого 386-е машины и их преемники не могли быть виртуализированы, следовательно, они не могли поддерживать гипервизор напрямую.

На самом деле ситуация была еще хуже, чем в этом кратком обзоре. Вдобавок к проблемам с инструкциями, которые, выполняясь в пользовательском режиме, вызывали системные прерывания, были еще и инструкции, которые могли считывать конфиден-

циальное состояние, не вызывая системных прерываний. Например, на процессорах x86 выпуска до 2005 года программа путем чтения селектора своего кодового сегмента могла определять, в каком режиме она выполняется, в пользовательском или в режиме ядра. Операционная система, выполнявшая такое действие и позволявшая обнаруживать, что она в данный момент находится в пользовательском режиме, могла на основе этой информации принимать неправильные решения.

Эта проблема была окончательно решена, когда в начале 2005 года Intel и AMD представили виртуализацию на своих центральных процессорах (Uhlig, 2005). На центральных процессорах Intel она называлась **технологией виртуализации** (Virtualization Technology (VT)), а на центральных процессорах AMD она называлась **безопасной виртуальной машиной** (Secure Virtual Machine (SVM)). Далее в общем смысле будет использоваться термин VT. Обе технологии были вдохновлены примером работы IBM VM/370, но при этом имеют от нее небольшие отличия. Основной замысел заключался в создании контейнеров, в которых могли бы запускаться виртуальные машины. При запуске в контейнере гостевой операционной системы она продолжает работать в нем, пока ею не будет вызвано исключение и не будет осуществлено системное прерывание в гипервизоре, например, при выполнении инструкции ввода-вывода. Набор операций, вызывающих это системное прерывание, контролируется битовым массивом, устанавливаемым гипервизором. С таким расширением появилась возможность создания классической виртуальной машины, работающей по принципу вызова системного прерывания и эмуляции.

Дотошный читатель, наверное, заметил явное противоречие в предложенном до сих пор описании. С одной стороны, было сказано, что x86-машины не могли быть виртуализированы вплоть до появления архитектурного расширения, представленного в 2005 году, а с другой — было сказано, что VMware запустила свой первый гипервизор на x86 в 1999 году. Как все это вместе может быть правдой? Ответ заключается в том, что гипервизоры до 2005 года фактически не запускали настоящие гостевые операционные системы. Вместо этого они *переписывали* часть кода на лету, заменяя проблемные инструкции безопасной кодовой последовательностью, эмулирующей исходную инструкцию. Предположим, к примеру, что гостевая операционная система выполняет привилегированную инструкцию ввода-вывода или модифицирует один из привилегированных управляющих регистров центрального процессора (например, регистр CR3, содержащий указатель на каталог страниц). Важно, чтобы последствия выполнения таких инструкций ограничивались данной виртуальной машиной и не оказывали влияния на другие виртуальные машины или на сам гипервизор. Таким образом, небезопасные инструкции ввода-вывода заменялись системным прерыванием, которое после проверки безопасности выполняло эквивалентную инструкцию и возвращало результат. Поскольку происходила перезапись, этим трюком можно было воспользоваться для замены инструкций, являвшихся служебными, но не входивших в число привилегированных. Все остальные инструкции выполнялись обычным порядком. Эта технология известна как двоичная трансляция и более подробно будет рассмотрена в разделе 7.4.

Переписывать абсолютно все служебные инструкции нет необходимости. В частности, пользовательские процессы на гостевой операционной системе могут, как правило, выполняться без модификации. Если инструкция не входит в число привилегированных, но относится к служебным и ведет себя в пользовательских процессах не так, как в процессах ядра, то все нормально. Она все равно запускается

в пользовательской среде. Что же касается служебных инструкций, входящих в состав привилегированных, то можно, как обычно, прибегнуть к классической схеме с системным прерыванием и эмуляцией. Разумеется, VMM должен обеспечить получение соответствующих системных прерываний. Обычно в VMM имеется модуль, выполняемый в ядре и перенаправляющий системные прерывания к своим собственным обработчикам.

Другая форма виртуализации известна как **паравиртуализация**. Она сильно отличается от полной виртуализации, поскольку никогда даже не стремится представить виртуальную машину, которая бы выглядела как настоящее основное оборудование. Вместо этого она представляет машиноподобный программный интерфейс, который явно раскрывает факт наличия виртуальной среды. Например, он предлагает набор **гипервызовов** (hypercall), позволяющих гостю отправлять явные запросы к гипервизору (во многом это похоже на то, как системный вызов предлагает службы ядра приложениям). Гостевые системы используют гипервызовы для привилегированных служебных операций, таких как обновление таблиц страниц, но поскольку они делают это явным образом во взаимодействии с гипервизором, в целом система может получаться проще и быстрее.

Наверное, то, что в паравиртуализации нет ничего нового, уже не вызовет у вас удивления. Разработанная IBM операционная система VM уже предлагала такую возможность, правда, под другим именем, еще с 1972 года. Идея была возрождена в мониторах виртуальных машин Denali (Whitaker et al., 2002) и Xen (Barham et al., 2003). По сравнению с полной виртуализацией, недостатком паравиртуализации является то, что гостевая система должна знать о существовании API виртуальной машины. Как правило, это означает, что она должна быть специально настроена под гипервизор.

Перед тем как еще больше углубиться в рассмотрение гипервизоров первого и второго типов, важно отметить, что не все технологии виртуализации пытаются обмануть гостевую систему, заставляя ее поверить в обладание всей системой. Иногда цель заключается в простом разрешении запуска процесса, изначально написанного для другой операционной системы и/или архитектуры. Поэтому нужно различать полную систему виртуализации и **виртуализацию на уровне процесса**. Хотя далее в главе основное внимание будет уделено первой из них, практикуется также и технология виртуализации на уровне процесса. К широко известным примерам можно отнести уровень совместимости WINE, который позволяет приложениям Windows запускаться на POSIX-совместимых системах, таких как Linux, BSD и OS X, и версию эмулятора QEMU на уровне процесса, которая позволяет приложениям для одной архитектуры выполняться на оборудовании с другой архитектурой.

7.3. Гипервизоры первого и второго типа

В работе Goldberg (1972) различаются два подхода к виртуализации. Одна разновидность гипервизора, названная **гипервизором первого типа** (type 1 hypervisor), показана на рис. 7.1, *а*. Технически этот гипервизор похож на операционную систему, поскольку это единственная программа, запущенная в самом привилегированном режиме. Его работа заключается в поддержке нескольких копий имеющегося оборудования, которое называется виртуальными машинами, что похоже на выполнение процессов в обычной операционной системе.

В отличие от этого **гипервизор второго типа**, показанный на рис. 7.1, *б*, относится к другой разновидности программ. Это программа, которая при распределении и планировании использования ресурсов опирается, скажем, на Windows или Linux и очень похожа на обычный процесс. Разумеется, гипервизор второго типа к тому же притворяется полноценным компьютером с центральным процессором и различными устройствами. Оба типа гипервизоров должны выполнять набор машинных инструкций безопасным образом. Например, операционная система, запущенная поверх гипервизора, может изменять и даже портить собственные таблицы страниц, но не те таблицы, которые принадлежат другим системам.

Операционная система, запущенная поверх гипервизора, в обоих случаях называется **гостевой операционной системой** (guest operating system). В гипервизоре второго типа операционная система, которая запущена на оборудовании, называется **основной операционной системой** (host operating system) или хост-системой. Первым гипервизором второго типа на рынке x86 был **VMware Workstation** (Bugnion et al., 2012). В этом разделе будет представлен общий замысел, положенный в ее основу, а исследовать VMware предстоит в разделе 7.12.

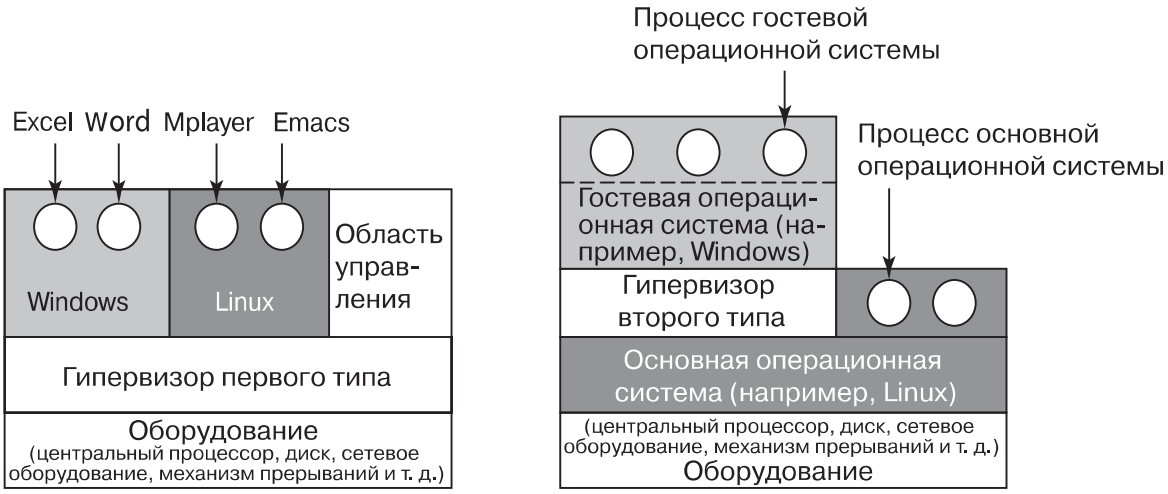

Рис. 7.1. Расположение гипервизоров: *а* — первого и *б* — второго типа

Многие функциональные возможности гипервизоров второго типа, которые иногда называют **гипервизорами, интегрированными с хост-системой** (hosted hypervisors), зависят от основной операционной системы, например от Windows, Linux или OS X. При первом запуске они ведут себя как только что загруженный компьютер и рассчитывают найти DVD, USB-накопитель или компакт-диск, содержащий операционную систему. Но в данном случае приводом может быть виртуальное устройство. Например, образ может быть сохранен как ISO-файл на жестком диске хост-системы, и гипервизор притворится, что чтение идет с надлежащего DVD-привода. Затем он устанавливает операционную систему на свой **виртуальный диск** (который в реальности опять является файлом Windows, Linux или OS X) путем запуска программы установки, найденной на DVD. После установки гостевой операционной системы на виртуальный диск ее можно будет загрузить и запустить.

Различные категории виртуализации для гипервизоров как первого, так и второго типа, о которых уже говорилось, сведены в табл. 7.1. Для каждой комбинации гипервизора и разновидности виртуализации приведены примеры.

Таблица 7.1. Примеры гипервизоров. Гипервизоры первого типа работают непосредственно на оборудовании, а гипервизоры второго типа используют службы существующей основной операционной системы

Метод виртуализации	**Гипервизор**	
	первого типа	**второго типа**
Виртуализация без поддержки оборудования	ESX Server 1.0	VMware Workstation 1
Паравиртуализация	Xen 1.0	—
Виртуализация с поддержкой оборудования	vSphere, Xen, Hyper-V	VMware Fusion, KVM, Parallels
Виртуализация на уровне процесса	—	Wine

7.4. Технологии эффективной виртуализации

Способность к виртуализации и производительность являются весьма важными вопросами, поэтому давайте их исследуем более подробно. Предположим, что в данный момент у нас есть гипервизор первого типа, поддерживающий одну виртуальную машину (рис. 7.2). Как и все другие гипервизоры первого типа, он работает непосредственно на оборудовании. Виртуальная машина запущена как пользовательский процесс в пользовательском режиме, и как таковой ей нельзя выполнять служебные инструкции (в толковании Попека — Голдберга). Но на виртуальной машине работает гостевая операционная система, которая считает, что она запущена в режиме ядра (хотя, разумеется, это не так). Мы назовем это **виртуальным режимом ядра** (virtual kernel mode). На виртуальной машине также запущены пользовательские процессы, которые считают, что они выполняются в пользовательском режиме (и действительно находятся в этом режиме).

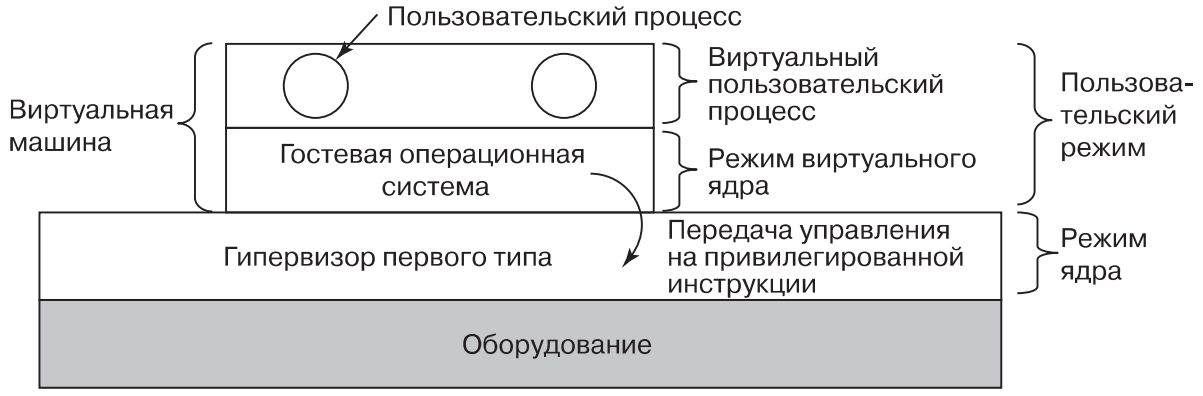

Рис. 7.2. Когда операционная система в виртуальной машине выполняет инструкцию, предназначенную только для режима ядра, при наличии технологии виртуализации она подвергается системному прерыванию и передает управление гипервизору

Что произойдет, когда гостевая операционная система (которая считает, что выполняется в режиме ядра) выполняет инструкцию, разрешенную, лишь когда центральный процессор реально работает в режиме ядра? Обычно на центральных процессорах без

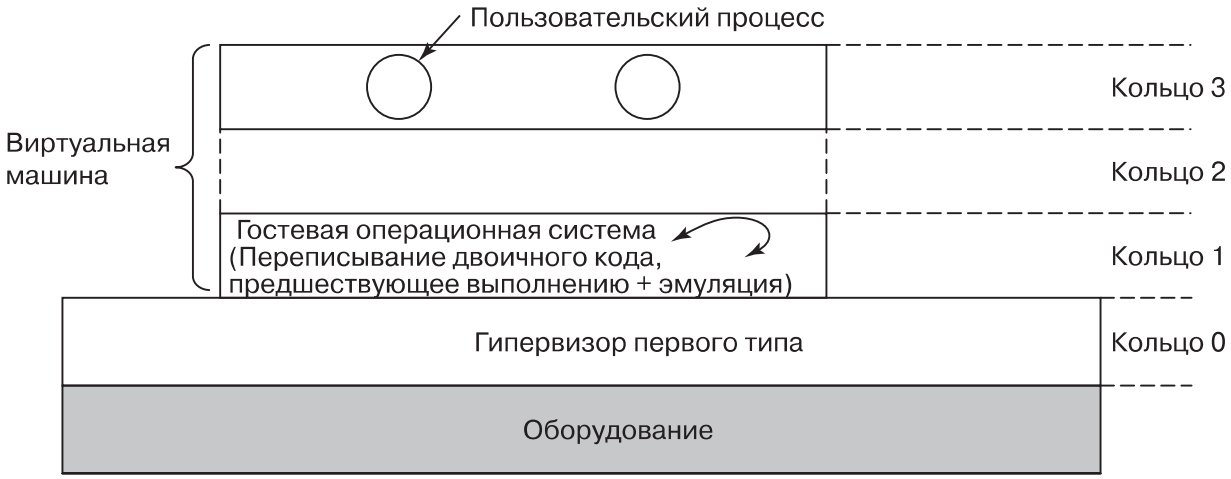

Рис. 7.3. Двоичный транслятор переписывает инструкции гостевой операционный системы, работающей в кольце 1, в то время как гипервизор работает в кольце 0

занимающейся их обработкой. Переход в последней инструкции также заменяется вызовом, направленным в гипервизор (чтобы гарантировать возможность повторения процедуры для следующего базового блока). Динамическая трансляция и эмуляция представляются весьма затратным делом, но обычно это не так. Транслируемые блоки кэшируются, что исключает их трансляцию в будущем. Кроме того, большинство блоков кода не содержат служебных или привилегированных инструкций и поэтому могут выполняться обычным образом. В частности, при условии, что гипервизор производит тщательную настройку аппаратного обеспечения (как это делает, к примеру, VMware), двоичный транслятор может игнорировать все пользовательские процессы. Они в любом случае выполняются в непривилегированном режиме.

После того как выполнение базового блока завершится, управление возвращается гипервизору, который затем находит его преемника. Если этот преемник уже был оттранслирован, он может быть выполнен без промедлений. В противном случае он сначала проходит трансляцию, кэшируется, а затем выполняется. В итоге основная часть программы попадет в кэш и будет выполняться практически на полной скорости. Используются различные оптимизации: например, если базовый блок завершается инструкцией перехода на другой базовый блок (или инструкцией его вызова), последняя инструкция может быть заменена переходом непосредственно на оттранслированный базовый блок, исключая все издержки, связанные с поиском блока-преемника. Опять же исключается необходимость замены служебных инструкций в пользовательских программах, оборудование в любом случае будет их просто игнорировать.

В то же время двоичная трансляция довольно часто применяется ко всему коду гостевой операционной системы, выполняемой в кольце 1, и при этом даже заменяются привилегированные служебные инструкции, что в принципе может быть сделано также при обработке служебного прерывания. Причина в том, что эти служебные прерывания обходятся очень дорого, а двоичная трансляция приводит к достижению более высокой производительности.

Все сказанное до сих пор относилось к гипервизорам первого типа. Хотя концептуально гипервизоры второго типа отличаются от гипервизоров первого типа, в целом в них используются те же самые технологии. Например, VMware ESX Server (гипервизор, впервые поставленный в 2001 году) использует в точности такую же двоичную транс-

у Linux). Возможно, в будущем API гипервизора или микроядра будет стандартизировано и последующие операционные системы будут спроектированы для вызова этого интерфейса вместо использования служебных инструкций. Тогда технология и использование виртуальных машин упростятся.

Разница между виртуализацией и паравиртуализацией показана на рис. 7.4. На нем изображены две виртуальные машины, поддерживаемые VT-оборудованием. На левой машине в качестве гостевой операционной системы используется немодифицированная версия Windows. При выполнении служебной инструкции оборудование инициирует системное прерывание с передачей управления гипервизору, который затем эмулирует эту инструкцию и возвращает управление. На правой машине используется версия Linux, модифицированная таким образом, что в ней больше не содержатся служебные инструкции. Вместо этого, когда у нее возникает потребность во вводе-выводе или внесении изменений в важные внутренние регистры (например, в регистр, указывающий на таблицы страниц), она для выполнения данной работы осуществляет вызов гипервизора точно так же, как прикладная программа совершает системный вызов в стандартной Linux.

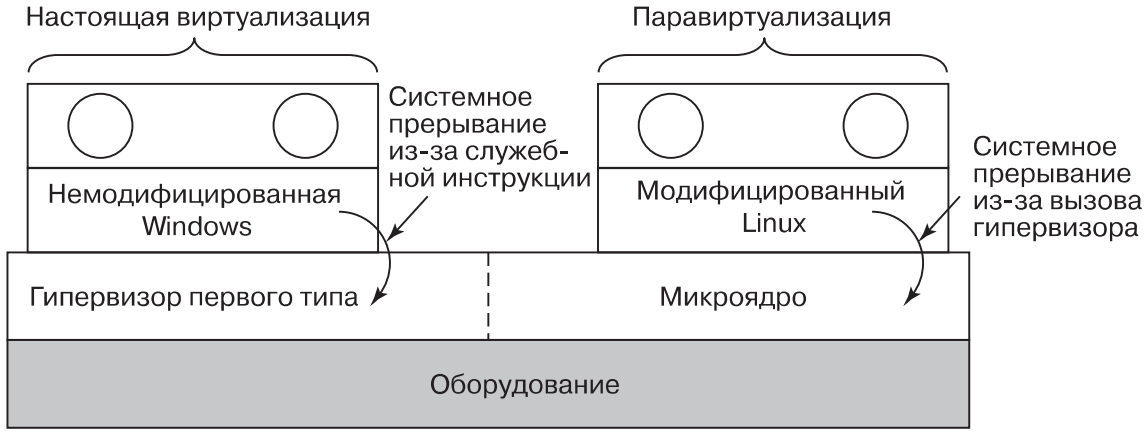

Рис. 7.4. Настоящая виртуализация и паравиртуализация

На рис. 7.4 показан гипервизор, разделенный на две части пунктирной линией. На самом деле на оборудовании выполняется только одна программа. Одна ее часть отвечает за интерпретацию перехваченных служебных инструкций, в данном случае от Windows. А другая часть просто выполняет гипервызовы. На этой части имеется надпись: «Микроядро». Если гипервизор предназначен для запуска только паравиртуализированных гостевых операционных систем, ему не нужно эмулировать служебные инструкции и мы имеем дело с настоящим микроядром, которое просто предоставляет самые основные службы, такие как диспетчеризация процессов и управление MMU. Граница между гипервизором первого типа и микроядром уже стирается и будет становиться все менее различимой по мере того, как гипервизоры будут приобретать все больше и больше функциональных возможностей и гипервызовов, что представляется вполне возможным развитием событий. Это спорная тема, но становится все более понятно, что программа, запущенная в режиме ядра непосредственно на оборудовании, должна быть невелика по размеру и надежна и состоять не из миллионов, а из тысяч строк кода.

По паравиртуализации гостевой операционной системы возникает ряд вопросов. Во-первых, если служебные инструкции заменены вызовами гипервизора, то как может

операционная система работать на простом оборудовании? Ведь оборудование не понимает эти гипервызовы. Во-вторых, что, если на рынке имеется несколько гипервизоров, например VMware, Xen с открытым кодом, изначально созданный в Кембриджском университете, и Hyper-V компании Microsoft, и у каждого из них имеется в чем-то отличающийся API-интерфейс гипервизора? Как можно модифицировать ядро для запуска всех этих гипервизоров?

Решение было предложено в работе Amsden et al. (2006). В их модели ядро, когда ему нужно выполнить какие-нибудь служебные инструкции, модифицируется для вызова специальных процедур. Все вместе эти процедуры, названные интерфейсом виртуальной машины (Virtual Machine Interface (**VMI**)), образуют низкоуровневый слой, который взаимодействует с оборудованием или гипервизором. Эти процедуры разработаны с прицелом на универсальность и не привязаны к какой-либо конкретной аппаратной платформе или к какому-нибудь конкретному гипервизору.

Экземпляр такой технологии для паравиртуализированной версии Linux, названной ими VMI Linux (VMIL), показан на рис. 7.5. Когда VMI Linux запускается непосредственно на оборудовании, она должна быть подключена к библиотеке, которая, как показано на рис. 7.5, *а*, выдает необходимые для работы настоящие (служебные) инструкции. При работе в режиме гипервизора, например VMware или Xen, гостевая операционная система подключается к другим библиотекам, совершающим соответствующие (и уже другие) гипервызовы, направляемые базовому гипервизору. Таким образом, ядро операционной системы сохраняет переносимость при успешно и эффективно взаимодействующем с ним гипервизоре.

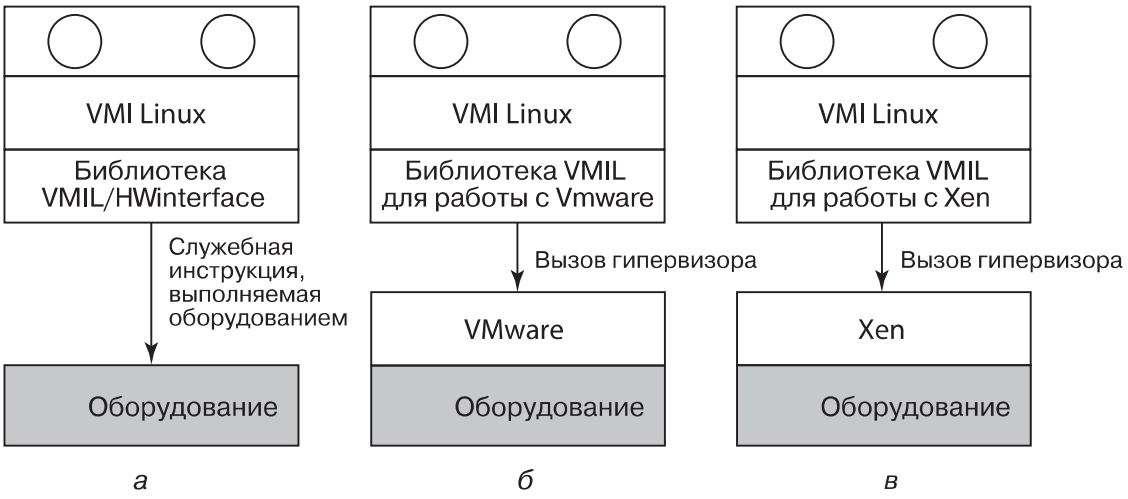

Рис. 7.5. VMI Linux, работающая: *а* — непосредственно на оборудовании; *б* — на VMware; *в* — на Xen

Относительно интерфейса виртуальной машины были внесены и другие предложения. Одно из популярных предложений называется **paravirt ops**. Концептуально идея похожа на то, что уже было описано ранее, но отличается в деталях. По сути, группа поставщиков Linux, включающая такие компании, как IBM, VMware, Xen и Red Hat, выступила в поддержку независимого от конкретного гипервизора интерфейса для Linux. Интерфейс, включенный в основную линейку ядра, начиная с версии 2.6.23, позволял ядру взаимодействовать с тем гипервизором, который управлял физическим оборудованием.

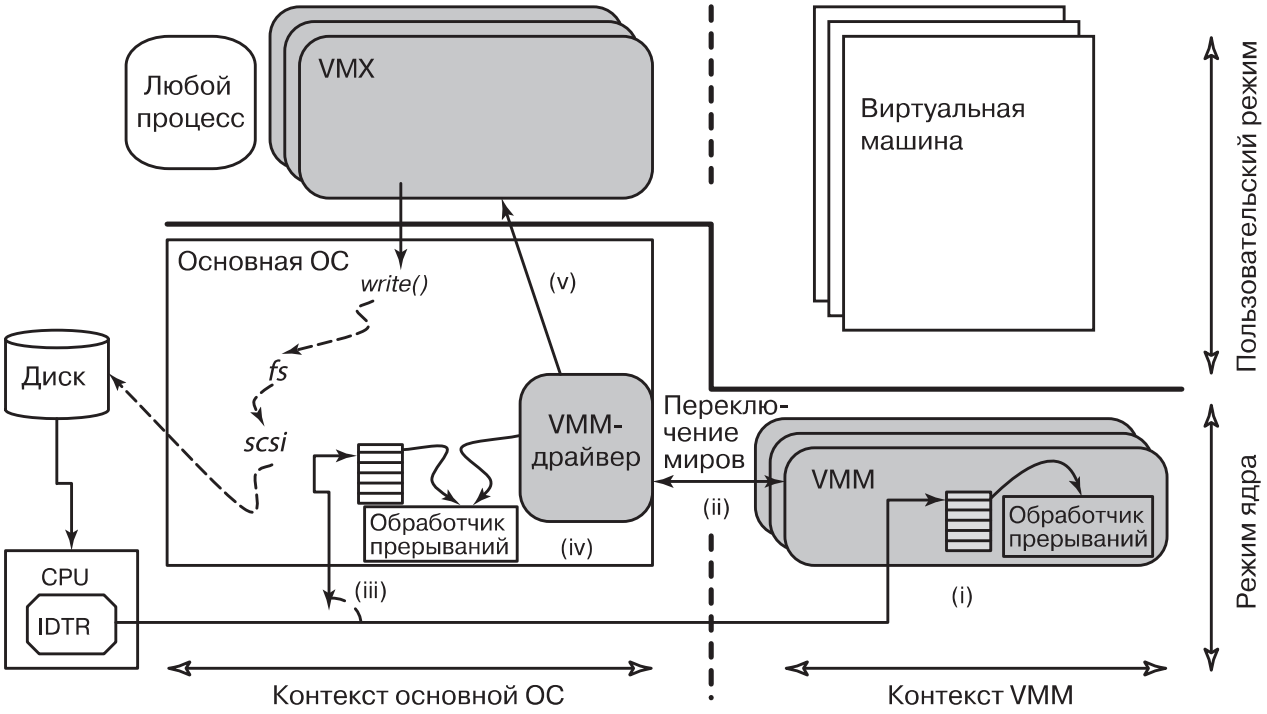

Рис. 7.8. VMware Hosted Architecture и три ее компонента: VMX, VMM-драйвер и VMM

- Небольшой драйвер устройства, выполняемый в режиме ядра (**VMX-драйвер**), устанавливаемый в основную операционную систему. Он используется в основном для получения возможности запуска VMM путем временной приостановки всей основной операционной системы. В операционную систему обычно во время начальной загрузки устанавливается один VMX-драйвер.
- VMM, включающий все программное обеспечение, необходимое для мультиплексирования центрального процессора и памяти, в том числе обработчики исключений, обработчики перехватов и эмуляции, двоичный транслятор и модуль теневой страничной организации памяти. VMM запускается в режиме ядра, но не работает в контексте основной операционной системы. Иными словами, он не может напрямую полагаться на службы, предлагаемые основной операционной системой, но он также не связан какими-либо правилами или соглашениями, декларируемыми основной операционной системой. Для каждой виртуальной машины имеется один экземпляр VMM, создаваемый при запуске виртуальной машины.

VMware Workstation выглядит так, будто запускается поверх существующей операционной системы, и, фактически ее компонент VMX запускается как процесс этой операционной системы. Но VMM работает на системном уровне, имея полный контроль над оборудованием и не испытывая никакой зависимости от основной операционной системы. На рис. 7.8 показаны взаимоотношения между объектами: два контекста (основной операционной системы и VMM) являются равноправными по отношению друг к другу, и у каждого есть компонент пользовательского уровня и компонент ядра. VMM при запуске (в правой половине рисунка) проводит переконфигурацию оборудования, обрабатывает все прерывания и исключения ввода-вывода и поэтому безопасным способом временно удаляет основную операционную систему из своей виртуальной памяти. Например, размещение таблицы прерываний настраивается внутри VMM путем назначения регистру IDTR нового адреса. И наоборот, когда работает основная

операционная система (левая половина рисунка), VMM и его виртуальная машина точно так же удаляются из ее виртуальной памяти.

Переход между этими двумя совершенно независимыми контекстами системного уровня называется **переключением миров** (world switch). Самим названием подчеркивается, что во время переключения миров все касающееся программного обеспечения изменяется в отличие от обычного переключения контекстов, реализуемого операционной системой. На рис. 7.9 показана разница между двумя видами переключений. Обычное переключение контекстов между процессами *А* и *Б* меняет местами пользовательскую часть адресного пространства и регистры двух процессов, но ряд важных системных ресурсов оставляет без изменений. Например, часть адресного пространства, принадлежащая ядру, идентична для всех процессов, также не изменяются обработчики исключений. В отличие от этого при переключении миров изменяется все: все адресное пространство, все обработчики исключений, привилегированные регистры и т. д. В частности, адресное пространство ядра основной операционной системы отображается только при работе в контексте основной операционной системы. После переключения миров в контекст VMM оно удаляется из адресного пространства целиком, освобождая пространство для работы как VMM, так и виртуальной машины. Хотя это может показаться довольно сложным процессом, его можно реализовать весьма эффективно и занять для его выполнения всего лишь 45 инструкций на языке машины x86.

Рис. 7.9. Разница между обычным переключением контекста и переключением миров

Внимательный читатель может удивиться: а как насчет адресного пространства ядра гостевой операционной системы? Ответ простой: оно является частью адресного пространства виртуальной машины и присутствует при работе в контексте VMM. Поэтому гостевая операционная система может использовать все адресное пространство, в частности те же места в виртуальной памяти, что и основная операционная система. При совпадении основной и гостевой операционных систем (например, обе Linux) именно так все и происходит. Разумеется, все это «просто работает», потому что есть два независимых контекста и между ними происходит переключение миров.

Затем тот же внимательный читатель может вновь удивиться: а как насчет области VMM, находящейся на верхушке адресного пространства? Как уже говорилось, это пространство резервируется самим VMM, и эта часть адресного пространства не может быть использована виртуальной машиной напрямую. К тому же эта небольшая