Прозрачный механизм удаленного обслуживания системных вызовов

Яковенко П.Н.
yak@ispras.ru

Аннотация. В статье рассматривается подход к удаленному обслуживанию системных вызовов, не требующий модификации кода пользовательского приложения и операционной системы. Использование технологии аппаратной виртуализации для перехвата системных вызовов, чтения их параметров и записи результатов позволяет делегировать обслуживание перехваченных системных вызовов другой системе: виртуальной, выполняющейся на этом же физическом компьютере, или даже другой машине в сети. Возможность предоставлять отдельным процессам контролируемый доступ к ресурсам, к которым сама операционная система доступа не имеет, позволяет эффективно решать некоторые задачи компьютерной безопасности.

1. Введение

Решение некоторых задач компьютерной безопасности основано на ограничении доступа программного кода к различного рода ресурсам, в частности сетевым ресурсам и файловой системе. Предоставив доступ к этим ресурсам только отдельным доверенным программам, можно гарантировать (при дополнительном контроле целостности этих программ) выполнение важных требований по безопасности, среди которых следует отметить предотвращение утечки критически важной информации через различные каналы передачи данных: сетевое соединение, переносные (USB) накопители и др.

Современные операционные системы (ОС) предоставляют широкие возможности по управлению доступом процессов к ресурсам компьютера. Однако, недостаточная надежность массовых ОС (таких как Windows, Linux и др.) [1] делает задачу разработки независимых от ОС (ортогональных к ней) программных средств защиты информации актуальной. Такие средства защиты могут быть реализованы с использованием технологии аппаратной виртуализации, когда защищаемая система выполняется в аппаратной виртуальной машине (ВМ), а система защиты размещается в теле монитора виртуальных машин (также называемого гипервизором) [2,4]. Функционирование гипервизора на более высоком аппаратном уровне привилегий позволяет полностью контролировать выполнение, как кода ОС,

так и пользовательских программ, оставаясь при этом аппаратно защищенным от вредоносного воздействия со стороны кода в ВМ, в т.ч. кода, выполняющегося в привилегированном режиме.

Пользовательский процесс в современных ОС не имеет прямого доступа к аппаратным ресурсам; операционная система представляет процессу некоторую абстрактную модель аппаратного обеспечения, взаимодействие с которой осуществляется посредством набора операций – системных вызовов (СВ). В частности, для установления сетевого соединения с удаленным компьютером и передачи ему данных процессу необходимо выполнить в заданном порядке несколько вполне определенных системных вызовов (*socket*, *connect*, *send* и т.п.).

В наших работах [2,3] было показано, как при помощи разделения полномочий по обслуживанию ресурсов между виртуальными машинами и делегирования обслуживания системных вызовов от одной виртуальной машины другой могут быть решены некоторые задачи компьютерной безопасности. Предложенные решения в целом базируются на схеме, изображенной на рисунке 1. Гипервизор обеспечивает одновременное выполнение двух, изолированных друг от друга, виртуальных машин. Обе ВМ работают под управлением одинаковой операционной системы. Первая ВМ – *вычислительная* – является основной. Пользователь может работать с ней в диалоговом режиме.

Оборудование, через которое осуществляется доступ к контролируемым ресурсам (ресурсы сети Интернет на рис. 1), физически отключается гипервизором от вычислительной ВМ. Это оборудование управляется другой – *сервисной* – виртуальной машиной, которая, вообще говоря, может выполняться скрытым для пользователя образом в фоновом режиме. Системные вызовы отдельных (доверенных) процессов перехватываются гипервизором и те из них, которые относятся к контролируемым ресурсам, передаются на обслуживание (обслуживаются удаленно) в сервисную ВМ. Заметим, что обслуживание этих вызовов внутри вычислительной ВМ неминуемо приведет к ошибке в силу отсутствия у нее возможностей (оборудования) осуществить доступ к соответствующим ресурсам. Остальные системные вызовы доверенных процессов, а также все системные вызовы других процессов, обслуживаются средствами ОС в вычислительной ВМ.

При перехвате системного вызова гипервизор может проконтролировать допустимость контекста, из которого выполняется запрос на системный вызов, т.е. разрешен ли данному процессу доступ к такой категории ресурсов, и выполнять удаленное обслуживание вызова только в случае успешного прохождения проверки. Анализируя параметры вызова, гипервизор может также осуществлять более тонкий контроль доступа к ресурсу, например, разрешать сетевой доступ только к ограниченному числу компьютеров в сети, имеющих заданные адреса.

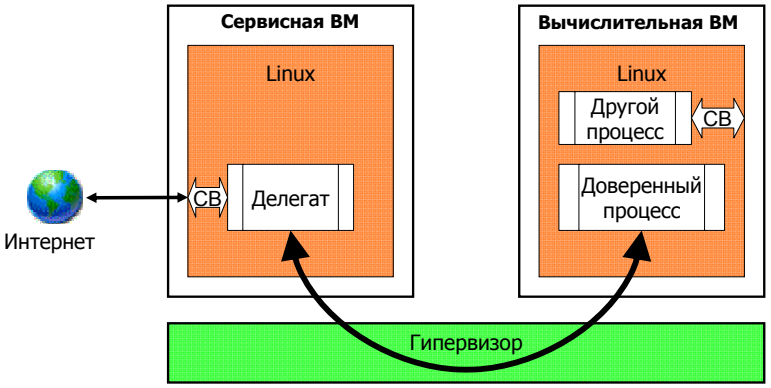

Рис. 1. Делегирование системных вызовов другой виртуальной машине.

В данной работе описывается архитектура и детали реализации механизма удаленного выполнения системных вызовов, используемого в системе безопасности, которая представлена в работе [3]. В ней контролируемыми ресурсами являются исключительно ресурсы сети Интернет. Как следствие, реализация, рассматриваемая в данной работе, допускает удаленное обслуживание только тех системных вызовов, которые могут быть использованы при сетевом взаимодействии через сокеты. Однако, учитывая общность представления сокетов и файлов для пользовательского процесса в виде дескрипторов файлов и, как следствие, общность многих системных вызовов, рассматриваемый механизм в целом пригоден также для тех сценариев, когда контролируемым ресурсом является файловая система.

Отдельные компоненты системы, рассматриваемой в данной работе, могут также использоваться для эффективного решения тех задач, в которых требуется выполнять трассировку системных вызовов. Дело в том, что штатные механизмы трассировки в ОС Linux (*ptrace*) базируются на сигналах, посылаемых ядром ОС процессу-монитору (отладчику) при выполнении отлаживаемым процессом системного вызова, что приводит к частому переключению контекста процессов (монитора и трассируемого процесса). Кроме того, чтение данных из адресного пространства трассируемого процесса возможно только порциями по 4 байта. Размещение монитора процессов на уровне гипервизора позволяет устранить указанные ограничения.

Задача удаленного выполнения системных вызовов рассматривается также в рамках проекта VirtualSquare [6]. Отличие предлагаемого нами подхода состоит в использовании технологии аппаратной виртуализации для перехвата системных вызовов. Кроме того, в работе [6] подмножество системных вызовов, исполняемых удаленно, задается жестко, в то время как в нашем подходе решение об удаленном выполнении системного вызова принимается

исходя из дескриптора ресурса, к которому обращается процесс. В ряде случаев системный вызов надо одновременно выполнять в обеих системах (локальной и удаленной) с последующим объединением результатов, и этот вопрос также рассмотрен в данной работе.

Задача удаленного выполнения системных вызовов близка задаче удаленного выполнения процедур [7], широко применяемой в распределенных программных системах, в частности, в сетевой файловой системе NFS [8]. Принципиальное отличие предлагаемого нами решения состоит в отсутствии необходимости модификации (в т.ч. перекомпиляции) кода приложения и операционной системы для обслуживания системных вызовов в другой системе.

Статья организована следующим образом. В разделе 2 представлен краткий обзор технологии аппаратной виртуализации и изложена общая архитектура системы и ее компонент. В разделе 3 детально рассмотрены принципы работы системы и обработка различных сценариев доступа к ресурсам. В разделе 4 представлены результаты анализа производительности системы как на синтетических тестах, так и на реальных приложениях. В разделе 5 подводятся итоги работы.

2. Архитектура системы

Технология виртуализации позволяет выполнять ОС в аппаратной виртуальной машине (ВМ) под управлением сравнительно небольшой по размеру системной программы – монитора виртуальных машин (гипервизора) [5]. Аппаратная виртуальная машина представляет собой эффективный изолированный дубликат реальной машины, и выполнение в ней операционной системы не требует внесения каких-либо изменений в код ОС. Гипервизор полностью контролирует взаимодействие ОС в ВМ с оборудованием и может обеспечить надежную изоляцию ВМ, опираясь на аппаратные механизмы защиты.

Функционирование гипервизора и организация выполнения виртуальной машины во многом схожи с тем, как операционная система управляет выполнением пользовательских процессов. Гипервизор инициализирует системные структуры данных, необходимые оборудованию, и выполняет специальную инструкцию VMRUN реализующую запуск ВМ и передачу управления соответствующей инструкции кода ОС. При возникновении аппаратного события (прерывания) или при выполнении операционной системой привилегированной инструкции (в т.ч. системного вызова) выполнение ВМ прерывается, и управление передается гипервизору на следующую инструкцию после VMRUN. После обработки перехваченного события гипервизор возобновляет выполнение ВМ.

Архитектура системы выглядит следующим образом. Гипервизор реализует одновременное выполнение двух виртуальных машин (Рис. 1), работающих

под управлением одинаковой операционной системы Linux. Отметим, что операционные системы в виртуальных машинах могут быть разными, если гипервизор при этом обеспечивает необходимое преобразование системных вызовов между исходной и целевой операционными системами. В вычислительной ВМ выполняются процессы пользователя, среди которых выделен набор *доверенных* процессов. Гипервизор предоставляет доверенным процессам привилегию доступа к ресурсам, обслуживаемым ОС в сервисной ВМ (контролируемым ресурсам); другие процессы доступа к этим ресурсам не имеют. Доступ доверенных процессов к контролируемым ресурсам производится посредством перехвата их системных вызовов и, при необходимости, их перенаправления для обслуживания в сервисную ВМ.

Обе виртуальные машины выполняются асинхронно, и вычислительная ВМ не блокируется на время удаленного обслуживания системного вызова. В это время процесс, инициировавший системный вызов, находится в состоянии ожидания, а все остальные процессы в вычислительной ВМ продолжают выполняться нормальным образом. Более того, система допускает одновременное обслуживание нескольких удаленных системных вызовов, поступающих от разных доверенных процессов.

При перехвате запроса процесса на выполнение системного вызова гипервизор копирует все входные параметры вызова в собственную область памяти и передает запрос на обслуживание вызова в сервисную ВМ. Системный вызов может быть блокирующим (например, *read*), и время его выполнения в общем случае не ограничено. Для избегания блокировки всей вычислительной ВМ на время обслуживания вызова процесс, инициировавший системный вызов, переводится в состояние ожидания, позволяя другим процессам продолжить свое выполнение. При поступлении из сервисной ВМ результатов обслуживания вызова гипервизор прерывает выполнение вычислительной ВМ, выводит процесс из состояния ожидания и копирует результаты в его адресное пространство. Пребывание процесса в состоянии ожидания реализуется штатными средствами ОС в ВМ, не вмешиваясь в работу механизма управления процессами в операционной системе.

Внутри сервисной ВМ выполняется набор процессов – *делегатов*, являющихся экземплярами специализированной программы. Каждый из делегатов обслуживает запросы на системные вызовы отдельного доверенного процесса из вычислительной ВМ, причем иерархия делегатов в сервисной ВМ соответствует иерархии доверенных процессов в вычислительной ВМ. Новый делегат порождается каждый раз при создании нового доверенного процесса и уничтожается при завершении работы этого процесса.

Выполнение делегата состоит в циклическом исполнении системных вызовов, поступивших от соответствующего доверенного процесса, и извещении гипервизора о результатах (делегат, в отличие от доверенного процесса, знает о существовании гипервизора). Делегаты получают запрос на выполнение

системного вызова от специализированного процесса – *диспетчера* – через механизм межпроцессного взаимодействия (очередь сообщений). Все запросы, поступающие от гипервизора, проходят через процесс-диспетчер. Диспетчер отслеживает соответствия между идентификаторами доверенных процессов и идентификаторами делегатов и, получив запрос от гипервизора на обслуживание системного вызова некоторого доверенного процесса, перенаправляет его соответствующему делегату. Дальнейшее обслуживание запроса целиком производится делегатом без участия диспетчера. В частности, делегат самостоятельно выполняет доступ к хранилищу, извещает гипервизор о результатах системного вызова и т.д.

Иерархия каждого доверенного процесса восходит к служебному процессу, являющемуся экземпляром специальной пользовательской программы – *монитора*. Первый доверенный процесс всегда порождается монитором. Монитор, в свою очередь, не является доверенной программой. Задача монитора состоит в запуске нового (в т.ч. первого) доверенного процесса и отслеживании его состояния, а также состояния всех его дочерних процессов, часть из которых могут быть доверенными, а часть – нет.

Монитор реализован на базе стандартного интерфейса отладки *ptrace*. Монитор перехватывает события порождения и завершения процессов, в т.ч. аварийного, например, при получении процессом сигнала, для которого у него не зарегистрирован обработчик. При выполнении одним из дочерних процессов системного вызова *fork* или *exec* монитор определяет, требуется ли данному процессу выполнение в доверенном режиме (т.е. будет ли новый процесс доверенным), и, в случае необходимости, запрашивает гипервизор о его включении для процесса. При завершении доверенного процесса монитор также извещает об этом гипервизор.

```
[STARTER]
/usr/local/bin/scp
[ARGUMENTS]
myfile user@host:myfile
[TRUSTED_OBJECTS]
444:/usr/local/bin/ssh
444:ssh
[CANCEL_HANDLER]
444:0x8049af3
```

Рис. 2. Паспорт доверенной задачи.

Монитор определяет, для каких процессов запрашивать доверенный режим выполнения, основываясь на специальном конфигурационном файле – *паспорте* задачи (Рис. 2). Паспорт содержит имя изначально запускаемой программы (не обязательно доверенной) и список передаваемых ей параметров командной строки. Основная часть паспорта состоит из набора

шаблонов для идентификации новых процессов, для которых следует запрашивать включение доверенного режима. Для каждого шаблона указывается уникальный идентификатор доверенной программы, зарегистрированной в гипервизоре. В гипервизоре для каждой зарегистрированной программы перечислен набор хэш-кодов, позволяющих проверить, что запускаемая программа действительно является одной из доверенных [1].

При выполнении дочерним процессом системного вызова *exec* монитор производит поиск шаблона, который может быть сопоставлен имени запускаемой программы, и, в случае успеха, делает запрос гипервизору на включение доверенного режима, сообщая ему идентификатор процесса (PID) и идентификатор соответствующей доверенной программы (например, 444 на рис. 2). Гипервизор проверяет допустимость включения доверенного режима для процесса в данной точке выполнения и, в случае соблюдения контекстных условий безопасности [1], активирует доверенный режим. При перехвате системного вызова *fork* монитор активирует доверенный режим для дочернего процесса только в том случае, если родительский процесс выполняется в доверенном режиме. Следует отметить, что гипервизор контролирует корректность выполнения запроса, т.е. что родительский процесс выполняется в доверенном режиме и действительно выполнил системный вызов *fork*.

Паспорт задачи также содержит адрес произвольной RET инструкции в коде доверенной программы. Монитор при помощи модуля (расширения) ядра в вычислительной ВМ регистрирует для доверенного процесса обработчик сигнала, не используемого процессом, по этому адресу. Посылка такого сигнала доверенному процессу приведет просто к выполнению инструкции RET. Этот сигнал используется для отмены выполнения системного вызова в вычислительной ВМ в тех случаях, когда для корректного обслуживания системного вызова его требуется выполнять в обеих виртуальных машинах (см. раздел 3.1).

2.1. Компоненты системы и их взаимодействие

Удаленное обслуживание системных вызовов реализуется гипервизором совместно с другими компонентами системы, функционирующими в обеих ВМ – вычислительной и сервисной. Компоненты функционирует как в пользовательском пространстве (монитор, диспетчер, делегаты), так и в пространстве ядра ОС (загружаемые модули ядра ОС).

В ходе инициализации системы в ядро ОС в вычислительной и сервисной ВМ динамически загружаются модули ядра. Каждый модуль выделяет непрерывное пространство физической памяти (по умолчанию 1 страницу размером 4Кб) для организации кольцевого буфера, регистрирует несколько обработчиков прерываний, при помощи которых гипервизор извещает виртуальную машину о событиях, требующих обработки и сообщает эту информацию (адрес буфера и номера прерываний) гипервизору посредством

гипервызова. В сервисной ВМ также запускается пользовательский процесс – диспетчер.

В ходе удаленного обслуживания системного вызова компоненты системы взаимодействуют между собой, причем механизмы взаимодействия реализованы по-разному (рис. 3). Реализация механизма взаимодействия некоторой пары компонент определяется уровнями привилегий, на которых они выполняются. Любая компонента может обратиться к гипервизору посредством гипервызова. Выполнение ВМ при этом прерывается, и управление передается гипервизору. Синхронный характер этого обращения позволяет передавать параметры гипервызова аналогично тому, как пользовательский процесс передает параметры ядру ОС при выполнении системного вызова: числовые параметры и адреса областей памяти передаются через регистры, при необходимости гипервизор читает область памяти виртуальной машины по указанным адресам и извлекает из нее (или записывает в нее) дополнительную информацию.

Пользовательский процесс (диспетчер или монитор) обращается за сервисом к модулю ядра посредством системных вызовов. Модуль ядра регистрирует в ОС специальное логическое устройство, видимое на уровне файловой системы как файл. Операции доступа к этому файлу (системные вызовы *read*/*write*/*ioctl*) вызывают соответствующие функции в драйвере логического устройства. Драйвер обрабатывает запрос процесса и возвращает управление ему. Если операция блокирующая, то драйвер может приостановить выполнение процесса до тех пор, пока он не сможет обслужить запрос. Модуль ядра обращается к пользовательскому процессу посредством посылки сигналов.

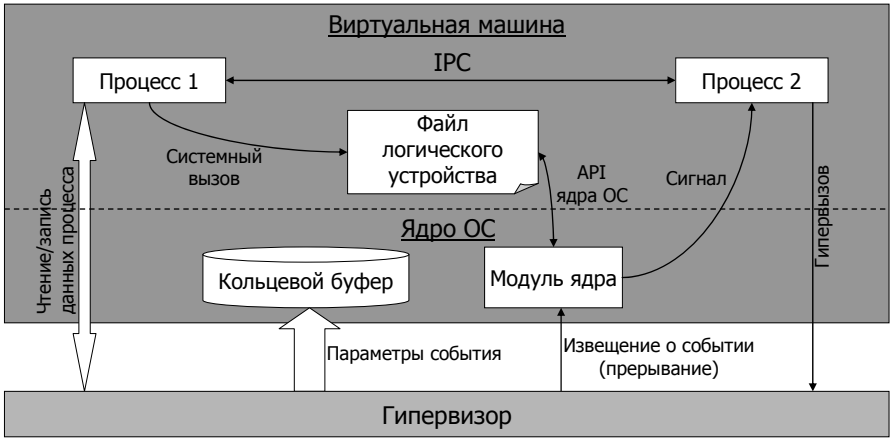

Рис. 3. Взаимодействие компонент системы.

Взаимодействие пользовательских процессов друг с другом (например, диспетчера с делегатами) осуществляется при помощи стандартных механизмов межпроцессного взаимодействия (IPC) ОС Linux – разделяемой памяти, очереди сообщений и пр. Все делегаты в сервисной ВМ являются членами одной иерархии процессов, восходящей к диспетчеру, что облегчает контроль создания разделяемых между процессами ресурсов: ресурс, создаваемый родителем, доступен потомку, а ресурс, создаваемый потомком, не доступен родителю.

Наиболее сложной является ситуация, в которой гипервизору требуется известить компоненты в виртуальной машине о некотором событии. Для этого гипервизор использует возможность, предоставляемую аппаратурой виртуализации, вбрасывать прерывания и исключительные ситуации в виртуальную машину посредством соответствующих полей в управляющей VMCB структуре ВМ. Тогда после возобновления ВМ аппаратура обеспечивает ей доставку прерывания непосредственно перед выполнением первой инструкции в ВМ. В результате вброса прерывания ОС передает управление на обработчик (вектор) данного прерывания, зарегистрированный модулем ядра в таблице обработчиков прерываний в процессе инициализации системы.

Параметры события передаются через кольцевой буфер. Буфер физически расположен в области памяти ВМ и разделяется между гипервизором и ВМ по схеме «поставщик - потребитель». Буфер представляет собой замкнутый в кольцо массив структур данных (фиксированного размера), голова которого сдвигается по мере выемки запросов из буфера, а хвост – по мере помещения запросов в буфер. Если буфер переполнен, то доставка запроса откладывается до тех пор пока в буфере не освободится место, т.е. пока ОС не обработает хотя бы одно из ранее сгенерированных событий. Запросы, ожидающие доставки в ВМ, накапливаются в очереди в памяти гипервизора.

Структура данных, представляющая собой элемент кольцевого буфера, едина для всех событий и включает поля для всех возможных параметров фиксированной длины. Параметры переменной длины передаются через отдельный буфер переменного размера, расположенный в памяти гипервизора – *хранилище*. Координаты параметра переменной длины – смещение от начала хранилища и длина, – специфицируются в структуре данных кольцевого буфера. Например, для системного вызова *write* структура включает 3 поля: идентификатор файлового дескриптора, начало (смещение) буфера в хранилище и длина буфера. Для каждого доверенного процесса гипервизор поддерживает отдельный экземпляр хранилища.

При получении запроса, содержащего параметры переменной длины, код в ВМ, которому предназначается этот запрос (например, делегат) выполняет гипервызов на доступ к хранилищу, передавая координаты запрашиваемого параметра и адрес буфера в собственной памяти, в который должны быть записаны данные из хранилища. Гипервизор обслуживает запрос и

возобновляет выполнение ВМ. При этом он контролирует, что границы запрашиваемого блока данных не выходят за пределы хранилища. Доступ к хранилищу возможен как по чтению, так и по записи.

При необходимости передать запрос одной из компонент внутри ВМ гипервизор ожидает, когда выполнение ВМ будет прервано по тому или иному событию (например, по таймеру) и анализирует, может ли он послать запрос в данной точке. Для этого в кольцевом буфере должно быть свободное место, а прерывания не должны быть маскированы в ВМ. Если это так, то гипервизор (при необходимости) заполняет хранилище параметрами переменной длины, формирует структура данных для кольцевого буфера, указывая в ней координаты параметров в хранилище, записывает сформированный запрос в буфер и вбрасывает прерывание. Вброс прерывания передает управление обработчику прерывания, который анализирует содержимое буфера и либо обслуживает его самостоятельно, либо передает запрос другой компоненте, отвечающей за обслуживание прерывания (например, диспетчеру).

Если получателем запроса гипервизора является пользовательский процесс (например, диспетчер), то доставка такого запроса производится транзитом через драйвер логического устройства. Пользовательский процесс обращается к файлу устройства и, в случае отсутствия запроса на данный момент, переходит в состояние ожидания. При поступлении запроса от гипервизора обработчик прерывания извещает об этом драйвер устройства. Тот, в свою очередь, читает запрос из кольцевого буфера, копирует его в память процесса и выводит его из состояния ожидания. Если запрос содержит параметры переменной длины, то доступ к хранилищу осуществляется тем пользовательским процессом, который непосредственно обслуживает запрос.

3. Прозрачное обслуживание системных вызовов

Механизм системных вызовов в процессорах семейства x86 может быть реализован различными способами. Исторически для этого использовались программные прерывания (инструкция INTn), в частности, ОС Linux использует 128 вектор для выполнения системного вызова. Возврат из системного вызова производился при помощи инструкции IRET. При выполнении инструкций INTn и IRET процессор проводит ряд проверок контекста выполнения инструкции и ее параметров. Частое выполнение процессом системных вызовов может оказывать существенное влияние на производительность системы. Как решение, производители процессоров предложили дополнительную пару инструкций специально предназначенную для быстрого перехода в режим ядра на заданный адрес и обратно: SYENTER/SYSEXIT от Intel и SYSCALL/SYSRET от AMD. Использование этих инструкций является предпочтительным, однако оригинальный механизм выполнения системных вызовов на базе программных прерываний по

прежнему поддерживается из соображений обратной совместимости приложений.

Рассматриваемая в этой работе система удаленного обслуживания системных вызовов требует, чтобы доверенные программы использовали механизм программных прерываний для выполнения системных вызовов. Это обусловлено возможностью перехвата инструкции программного прерывания и возврата из прерывания непосредственно при помощи аппаратуры виртуализации. Остальные процессы в вычислительной ВМ могут использовать произвольные механизмы системных вызовов. Из соображений повышения производительности перехват программного прерывания (инструкция INTn) устанавливается, только когда управление передается доверенному процессу, что позволяет предотвратить выход из ВМ, если эта инструкция выполнялась любым другим (недоверенным) процессом.

Флаг перехвата инструкции IRET, в свою очередь, устанавливается при каждом возврате управления ВМ, если в системе выполняется хотя бы один доверенный процесс. Лишь перехватывая все такие инструкции, гипервизор может отследить момент, когда ОС передает управление доверенному процессу. Это необходимо для корректного возобновления процесса после получения результатов из сервисной ВМ. Если результаты готовы и возврат управления происходит на следующую инструкцию после запроса на системный вызов, то гипервизор записывает результаты, полученные из сервисной ВМ, на регистры и в память процесса, и процесс продолжает выполнение.

При перехвате программного прерывания гипервизор проверяет, что оно было выполнено из контекста доверенного процесса, и что вектор прерывания соответствует вектору запросов на обслуживание системных вызовов (128 в ОС Linux). Далее гипервизор проверяет, требуется ли обслуживать данный системный вызов удаленно в сервисной ВМ или локально в вычислительной ВМ. Правила такого анализа системных вызовов будут рассмотрены в следующем разделе. Если вызов локальный, то гипервизор просто возобновляет управление ВМ, передавая управление ядру ОС. Если вызов удаленный, то гипервизор копирует параметры системного вызова из регистров и, возможно, из адресного пространства процесса в собственную область памяти, формирует запрос и отправляет его в сервисную ВМ. Схема механизма прозрачного обслуживания системных вызовов приведена на рисунке 4.

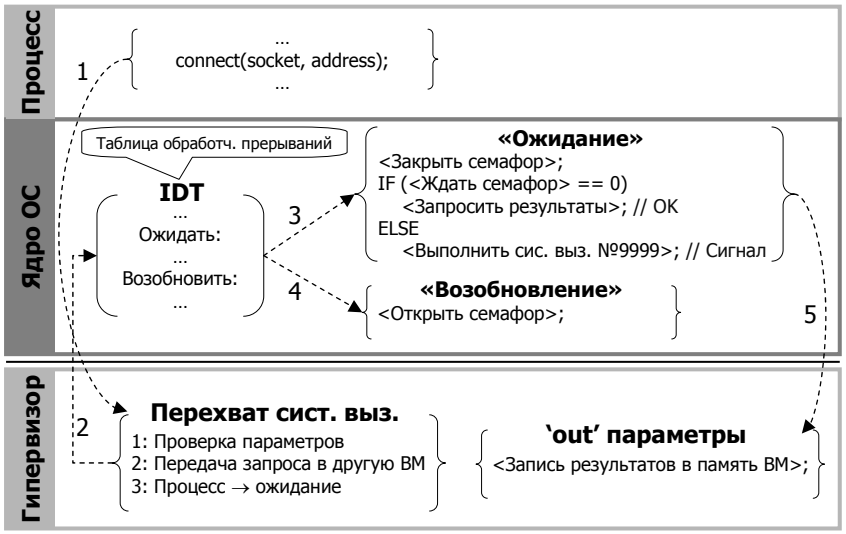

Рис. 4. Прозрачное обслуживание системного вызова в вычислительной ВМ.

Для определения адреса параметров системного вызова в физической памяти гипервизор программным образом обходит таблицы приписки процесса и вычисляет условно физический адрес в контексте ВМ. Далее, зная отображение памяти ВМ на машинную память, гипервизор вычисляет точное размещение параметров вызова в физической памяти. В процессе чтения параметров, расположенных в памяти процесса (например, в случае системного вызова *write*) возможна ситуация, когда данные расположены в странице памяти, откачанной ОС на внешнее устройство.

Гипервизор, обнаружив откачанную страницу, вбрасывает в ВМ исключение ошибка страницы с адресом, соответствующим отсутствующей странице, и возобновляет управление ВМ. ОС подкачивает страницу в память и возвращает управление процессу по адресу инструкции выполнения системного вызова. Процесс повторно выполняет системный вызов, и гипервизор заново начинает копирование параметров. Такой процесс будет повторяться до тех пор, пока все страницы памяти, занятые входными параметрами системного вызова, не окажутся в физической памяти.

После копирования входных параметров системного вызова гипервизор возобновляет выполнение вычислительной ВМ и переводит процесс в состояние ожидания. Для этого в точке выполнения системного вызова он вбрасывает синхронное прерывание, для которого модуль ядра в вычислительной ВМ зарегистрировал обработчик. В результате, вместо 128 прерывания, соответствующего системным вызовам, аппаратура доставляет другое прерывание.

Получив управление, обработчик осуществляет доступ к закрытому семафору, что переводит процесс в состояние ожидания штатными средствами ОС. Процесс ожидает открытие семафора в режиме, допускающем обработку внешних событий. В случае поступления сигнала для процесса ОС прерывает ожидание семафора. Модуль ядра при этом имитирует запрос на выполнение несуществующего системного вызова (например, с номером 9999). Ядро ОС, разумеется, не будет выполнять этот запрос, однако до того, как вернуть управление процессу, оно выполнит обработку поступивших сигналов.

После обслуживания сигнала ОС возвращает управление процессу на исходную инструкцию системного вызова, и процесс повторно выполняет запрос на системный вызов. Гипервизор перехватывает его, определяет, что в данное время системный вызов находится в процессе удаленного обслуживания, снова подменяет прерывание, и процесс во второй раз переходит в состояние ожидания. Такой циклический процесс будет повторяться до тех пор, пока из сервисной ВМ не поступят результаты выполнения системного вызова.

При получении результатов из сервисной ВМ гипервизору необходимо возобновить выполнение доверенного процесса со следующей инструкции, причем в регистр *r/eax* должен быть записан результат выполнения вызова, а в память процесса (например, в случае системного вызова *read*) по заданным адресам должны быть записаны выходные данные, если они имеются.

Гипервизор выводит процесс из состояния ожидания посредством вброса другого синхронного прерывания в вычислительную ВМ, обработчик которого освобождает семафор и, тем самым, возобновляет выполнение процесса. Процесс далее выполняет гипервызов на запрос результатов системного вызова. Заметим, что хотя гипервызов производится из контекста ядра, в точке гипервызова на аппаратуре загружены собственные таблицы приписки доверенного процесса. Гипервизор просматривает таблицы и проверяет, что виртуальные адреса, по которым должны быть записаны результаты отображены на физическую память и доступ к этим адресам открыт по записи. Наличие доступа только по чтению может означать, что данная область памяти является «копируемой при операции записи». При наличии доступа по записи ко всему необходимому диапазону адресов гипервизор копирует результаты выполнения системного вызова из хранилища в адресное пространство процесса и возобновляет выполнение ВМ. В противном случае гипервизор для всех необходимых виртуальных страниц вбрасывает в ВМ исключение «ошибка страницы», дающее ОС указание выделить память для соответствующей виртуальной страницы и отобразить ее на физическую память.

При удаленном обслуживании системного вызова делегат может получить сигнал от ОС в сервисной ВМ, например, сигнал SIGPIPE о разрыве сетевого соединения. В этом случае делегат извещает гипервизор о полученном сигнале, и гипервизор доставляет сигнал доверенному процессу. Для этого он

информирует о сигнале модуль ядра ОС, который, в свою очередь, доставляет сигнал процессу средствами API ядра ОС Linux. Доставка процессу сигнала может привести к его аварийному завершению. В этом случае монитор известит гипервизор об окончании выполнения процесса, и гипервизор освободит память, отведенную под служебные структуры данных.

3.1. Точка обслуживания системного вызова

В подавляющем большинстве случаев системный вызов может быть ассоциирован с ресурсами какой-либо одной из виртуальных машин либо на основании номера системного вызова, либо на основании его параметров. В частности, системный вызов *socket*, создающий конечную точку для сетевого взаимодействия и возвращающий файловый дескриптор для работы с ней, должен обслуживаться в сервисной ВМ. Системный вызов *write* требует дополнительного анализа файлового дескриптора, передаваемого в параметрах вызова. Если дескриптор ассоциирован с сокетом, то он должен обслуживаться в сервисной ВМ, в противном случае – в вычислительной ВМ.

ОС в обеих ВМ поддерживают наборы файловых дескрипторов соответственно для доверенного процесса и его делегата независимо друг от друга. Если гипервизор после удаленного обслуживания системного вызова вернет доверенному процессу идентификатор файлового дескриптора, полученного от делегата, без каких-либо изменений, то это может привести к наличию в памяти доверенного процесса двух идентичных дескрипторов, которые на самом деле соответствуют разным ресурсам в той и другой ВМ. В дальнейшем при выполнении доверенным процессом системного вызова для такого дескриптора гипервизор не сможет различить, относится ли вызов к локальному ресурсу в вычислительной ВМ или удаленному в сервисной ВМ.

Для решения этой проблемы гипервизор реализует дополнительный уровень абстракции файловых дескрипторов для доверенных процессов. Файловые дескрипторы, хранимые в памяти доверенного процесса (в каких-то переменных), представляют собой не реальные дескрипторы, назначенные ядром ОС в вычислительной или сервисной ВМ, а индексы в *таблице удаленных ресурсов*, поддерживаемой гипервизором. Гипервизор перехватывает и обрабатывает все системные вызовы доверенного процесса, у которых входные или выходные параметры содержать файловые дескрипторы. Если параметр входной, то гипервизор извлекает из таблицы удаленных ресурсов фактический файловый дескриптор, модифицирует параметры системного вызова и передает его на обслуживание той ВМ, которая является владельцем ресурса с этим дескриптором. Если параметр выходной, то гипервизор создает новую запись в таблице удаленных ресурсов, помечая, какая из виртуальных машин является владельцем ресурса, и модифицирует выходные параметры процесса, подставляя индекс созданной записи вместо фактического значения дескриптора. Компонента гипервизора, отвечающая за обработку файловых дескрипторов, учитывает особенности

выделения свободных дескрипторов ОС Linux, включая нюансы работы системных вызовов *dup2* и *fcntl*. Таким образом, по значению, передаваемому доверенным процессом всегда можно определить виртуальную машину – владельца ресурса и номер файлового дескриптора в ее контексте.

В ряде случаев выполнение системного вызова может задействовать ресурсы обеих ВМ. В частности, параметры системного вызова *select* и его аналогов, а именно наборы файловых дескрипторов, для которых процесс ожидает возникновения соответствующих событий, могут одновременно относиться как к ресурсам вычислительной, так и сервисной ВМ. Такой вызов должен обслуживаться одновременно в обеих ВМ, иначе программа может перестать выполняться корректно. При этом системный вызов в каждой виртуальной машине должен обслуживаться только с теми параметрами (файловыми дескрипторами), которые относятся к ресурсам данной ВМ.

При перехвате запроса на выполнение системного вызова, требующего одновременного выполнения в обеих ВМ, гипервизор, используя таблицу удаленных ресурсов, расшивает фактические параметры на два непересекающихся набора (по одному для каждой из ВМ) и выполняет вызов одновременно в обеих ВМ с соответствующими наборами параметров. Модифицированный набор параметров записывается в память доверенного процесса поверх оригинального. Перед возвратом управления процессу (после обслуживания системного вызова) гипервизор восстанавливает исходный набор параметров, возможно, корректируя его с учетом результатов системного вызова, полученных из сервисной ВМ. Итоговым результатом выполнения системного вызова является результат той ВМ, которая хронологически первой закончила выполнение своей части вызова, результаты другой ВМ отбрасываются.

По получении результатов от одной из ВМ гипервизор производит отмену выполнения системного вызова в другой ВМ. Механизм отмены выполнения системного вызова реализован в обеих виртуальных машинах по-разному. В случае вычислительной ВМ модуль ядра посылает доверенному процессу определенный сигнал (не используемый процессом). При этом модуль системы защиты непосредственно перед посылкой сигнала регистрирует для процесса специальный «пустой» обработчик сигнала, представляющий собой адрес RET инструкции в коде доверенной программы. Адрес инструкции указывается в паспорте задачи. Регистрация обработчика гарантирует, что посылка сигнала не приведет к аварийному останову процесса. В сервисной ВМ все системные вызовы, которые могут выполняться одновременно в обеих ВМ, выполняются в отдельном потоке (нити) делегата. Отмена выполнения системного вызова производится посредством принудительного завершения этого потока.

4. Производительность системы

Описанная в этой работе система реализована на базе монитора виртуальных машин KVM [9]. KVM включен в основную ветку разработки ядра ОС Linux и представляет собой модуль, динамически загружаемый в ядро базовой (хост) операционной системы Linux. Управление выполнением ВМ реализуется совместно ядром хост системы, модулем KVM и пользовательской программой QEMU. QEMU виртуализирует периферийные устройства и обеспечивает совместный доступ виртуальных машин к оборудованию, установленному на компьютере и управляемому базовой системой.

Реализация, представленная в этой работе, построена на базовой операционной системе Linux с ядром версии 2.6.31.6 и мониторе виртуальных KVM версии 88. Суммарный объем кода компонент системы составляет порядка 16 тыс. строк. Виртуальные машины выполняются под управлением ОС Linux из дистрибутива Fedora версии 9 со штатным ядром версии 2.6.27.12-78.2.8.fc9.i686. На компьютере установлен четырехядерный процессор Phenom 9750 компании AMD с тактовой частотой 2.4 Ггц и 2 Гбайта оперативной памяти. Данный процессор поддерживает технологию аппаратной виртуализации, включая виртуализацию памяти на базе вложенных (NPT) таблиц приписки виртуальной машины. Базовая операционная система использует все четыре ядра процессора (ядро базовой ОС собрано в SMP конфигурации). Каждой виртуальной машине выделяется по одному виртуальному процессору и по 512 Мбайт оперативной памяти.

Для проведения ряда тестов используется второй компьютер такой же конфигурации. В обоих компьютерах установлены 100Мбит-ные сетевые адаптеры, связанные друг с другом через сетевой концентратор (хаб). К концентратору подключены только данные две машины.

Доступ виртуальной машины к сети осуществляется посредством создания в базовой ОС штатными средствами ядра ОС программного сетевого интерфейса (TAP0). Этот интерфейс является образом сетевого интерфейса (ETH0) виртуальной машины в базовой системе. Привязка интерфейса TAP0 к физической среде производится через программный Ethernet мост (bridge) в ядре базовой ОС, также организуемый штатными средствами ОС Linux. Такая конфигурация (рис. 1) позволяет открыть ВМ для других машин в сети, в отличие от конфигурации QEMU по умолчанию, скрывающей ВМ от других машин в сети посредством механизма трансляции сетевых адресов (NAT) и разрешающей только исходящие соединения в ВМ.

Для тестирования производительности мы использовали четыре специально разработанных синтетических теста, моделирующих «тяжелые» для системы сценарии использования, и три типовых программы: утилиту тестирования производительности сети TTCP, программу удаленного доступа SSH и веб-сервер Apache.

Синтетические тесты основаны на выполнении системного вызова *select()* с одним или двумя файловыми дескрипторами. Один из дескрипторов (*локальный*), обозначим его *LocalFD*, представляет собой файл, открытый в контексте вычислительной ВМ, другой (*удаленный*), обозначим его *RemoteFD*, – сокет, созданный в сервисной ВМ. Во всех тестах, кроме первого, выполнение системного вызова требует взаимодействия между ВМ. Тесты запускаются из контекста вычислительной ВМ.

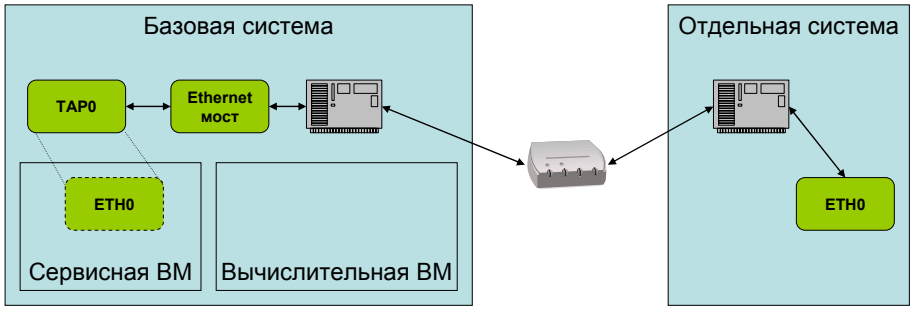

Рис. 5. Конфигурация сети для тестирования производительности.

Обрамление дескриптора квадратными скобками означает, что запрашиваемая операция не может быть выполнена для данного ресурса, и системный вызов заблокирует выполнение соответствующего пользовательского процесса в одной из ВМ. Отсутствие квадратных скобок означает возможность выполнения операции и немедленный возврат управления пользовательскому процессу. Тогда синтетические тесты проверяют следующие сценарии выполнения системного вызова:

- select(LocalFD);
- select(RemoteFD);
- select(LocalFD, [RemoteFD]);
- select([LocalFD], RemoteFD).

Выполнение системного вызова *select(LocalFD)* не требует взаимодействия между виртуальными машинами и характеризует «вычислительные» накладные расходы внутри вычислительной ВМ на перехват системных вызовов, анализ фактических параметров и пр. Выполнение системного вызова *select(RemoteFD)* показывает суммарное время, необходимое на доставку запроса делегату в сервисную ВМ, выполнение системного вызова в ней и возврат результатов процессу, пребывающему все это время в состоянии ожидания.

Выполнение системного вызова *select* с двумя дескрипторами включает в себя взаимодействие между виртуальными машинами и, кроме того, требует

выполнения отмены системного вызова в той ВМ, в которой процесс был заблокирован, а именно в той ВМ, дескриптор ресурса которой обрамлен квадратными скобками. Следует отметить принципиальное отличие третьего и четвертого теста. В тесте *select(LocalFD, [RemoteFD])* отмена системного вызова, выполняемого делегатом в сервисной ВМ, производится *асинхронно* для процесса в вычислительной ВМ, и он может продолжить свое выполнение, не дожидаясь подтверждения от сетевой ВМ. Такая оптимизация возможна, поскольку для нормального продолжения выполнения процесса достаточно результатов получаемых локально от ядра вычислительной ВМ. В свою очередь, в тесте *select([LocalFD], RemoteFD)* процесс не может продолжить выполнение, пока выполнение системного вызова не будет прервано, что приводит к дополнительным накладным расходам.

В таблице 1 указано время выполнения тестов (в секундах) в цикле из 100 тысяч итераций. Первая строка таблицы характеризуют выполнение теста *select(LocalFD)* в базовой системе. Вторая строка показывает время выполнения теста *select(LocalFD)* в ВМ со включенным механизмом отслеживания выполнения процесса. Возрастание времени выполнения на один порядок обусловлено затратами на перехват инструкции INTn, инициирующей системный вызов и, главное, инструкции IRET, реализующей возврат в пользовательский режим как из системного вызова, так из обработчиков прерываний. Мы ожидаем, что, адаптировав предлагаемую систему под механизм быстрого выполнения системных вызовов (SYSCALL/SYSRET), поддерживаемый современными процессорами семейства x86, данный показатель может быть улучшен.

Показатель в третьей строке таблицы говорит о том, что собственно вычислительные затраты системы (за исключением перехвата двух инструкций) составляют порядка 20%. Четвертая строка характеризует накладных расходы на асинхронную отмену части системного вызова, выполняемой делегатом в сервисной ВМ. Отметим, что несколько последовательных операций отмены также выполняются асинхронно, и синхронизация производится только при последующем выполнении «существенного» (не отменяемого) удаленного системного вызова. Кроме того, делегат не будет исполнять системный вызов, если обнаружит, что для него уже поступила команда на отмену. Такое возможно в силу асинхронности выполнения виртуальных машин.

В тесте select(LocalFD, [RemoteFD]) синхронизация производится только после выходе из цикла при закрытии «удаленного» сокета (системный вызов close). При наличии большого количества последовательно отмененных системных вызовов (100 тысяч в данном случае) такая операция синхронизации может занимать продолжительное время, что и отражено в таблице. Непосредственно при выходе из основного цикла время выполнения теста составляет всего 15 секунд. Последующая операция закрытия сокета,

требующая синхронизации операций отмены требует дополнительных 16 секунд.

Последние две строки таблицы характеризуют полноценное удаленное обслуживание системного вызова. Шестая строка таблицы показывает накладные расходы на отмену локальной части системного вызова в вычислительной ВМ, которая всегда выполняется синхронно для процесса.

Тест	Время (сек.)
Базовая система	1
Виртуальная машина	9
select(LocalFD)	11
select(LocalFD, [RemoteFD])	15 (31)
select(RemoteFD)	189
select([LocalFD], RemoteFD)	253

Таб. 1. Время выполнения синтетических тестов.

Синтетические тесты показывают, что при худшем сценарии время выполнения системного вызова может возрастать до 250 раз. Однако, на практике это не оказывает существенного влияния на время выполнения программы в целом. Во-первых, большинство системных вызовов, требующих удаленное выполнение, включает в себя передачу данных через периферийное устройство (в частности, по сети). Во-вторых, приложение, как правило, выполняет также другие действия, которые нивелируют данные накладные расходы, например, оно может часто находится в состоянии ожидания открытия семафора.

На рисунке 6 приведены результаты тестирования системы на утилите TTCP, выполняющей в цикле передачу пакетов между двумя машинами в сети. Первая диаграмма соответствует оригинальной TTCP утилите, вторая – модифицированной, в которую мы добавили выполнение системного вызова *select* перед каждой посылкой пакета. Системный вызов *select* в данном случае выполняется удаленно, т.е. соответствует синтетическому тесту select(RemoteFD). При выполнении оригинальной утилиты накладные расходы на удаленное обслуживание системных вызовов составили всего 2% от суммарного времени выполнения программы. Добавление системного вызова *select* увеличило накладные расходы до 31%, однако даже в этом случае они значительно меньше издержек в синтетическом тесте select(RemoteFD), где время выполнения увеличилось в 189 раз.

Мы также тестировали предлагаемую систему на утилите удаленного доступа SSH, посредством копирования файла между двумя машинами в сети, а также на веб-сервере Apache, запуская для него пакет тестов нагрузочного

тестирования Flood. В обоих случаях время выполнения тестов варьировалось в пределах 1% от их выполнения на базовой системе. Таким образом, мы считаем, что предлагаемый механизм удаленного обслуживания системных вызовов является достаточно эффективным для его использования в промышленных задачах.

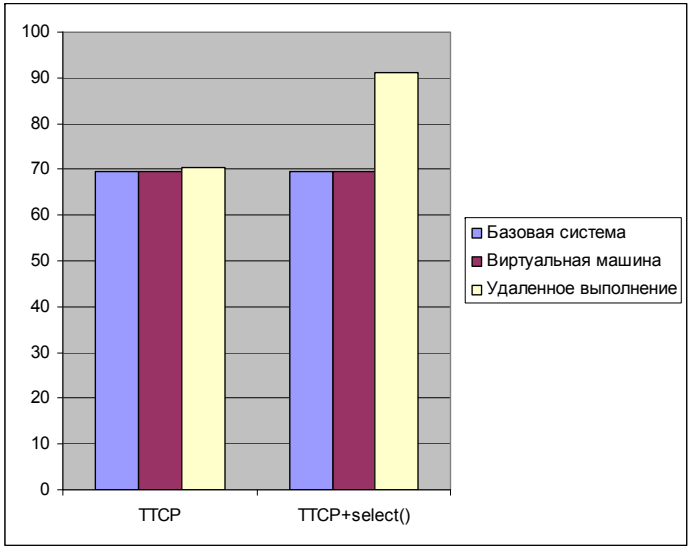

Рис. 6. Время выполнения утилиты TTCP (сек.).

5. Заключение

В данной работе представлен подход к удаленному выполнению системных вызовов пользовательского процесса, не требующий внесения каких-либо изменений (в т.ч. перекомпиляции) ни в код процесса, ни в код операционной системы. Особенностью рассматриваемого подхода является то, что процессу может быть предоставлен контролируемый доступ к таким ресурсам, к которым операционная система, под управлением которой выполняется процесс, ни прямого, ни опосредованного доступа не имеет. Это позволяет предоставлять селективный доступ к ресурсам, требующим контроля (например, сети Интернет), отдельным доверенным пользовательским процессам, в то же время оставляя эти ресурсы недоступными другим процессам и даже ядру операционной системы.

Предлагаемый подход основан на выполнении операционной системы (и управляемых ею процессов) внутри аппаратной виртуальной машины. Монитор виртуальных машин (гипервизор) перехватывает системные вызовы отдельных доверенных процессов и при необходимости передает их на

обслуживание сервисной машине. Сервисная машина может быть другой виртуальной машиной, выполняющейся параллельно, или другой физической машиной, с которой у гипервизора есть канал связи.

Рассматриваемый подход реализован на базе монитора виртуальных машин KVM. В качестве сервисной машины использовалась параллельно выполняющаяся виртуальная машина, а в качестве контролируемых ресурсов было выбраны сетевые ресурсы (в т.ч. ресурсы Интернета). Тестирование производительности системы показало, что на реальных приложениях накладные расходы на удаленное обслуживание системных вызовов укладываются в пределы 3 процентов.

Литература

[1] Tanenbaum, A. S., Herder, J. N., Bos, H. Can We Make Operating Systems Reliable and Secure?. Computer 39, 5 (May 2006), pp. 44-51.

[2] Burdonov, I., Kosachev, A., Iakovenko, P. Virtualization-based separation of privilege: working with sensitive data in untrusted environment. In Proceedings of the 1st Eurosys Workshop on Virtualization Technology for Dependable Systems, New York, NY, USA, 2009, ACM, pp. 1-6.

[3] Яковенко П.Н. Контроль доступа процессов к сетевым ресурсам на базе аппаратной виртуализации. Методы и средства обработки информации. Труды Третьей Всероссийской научной конференции, М, 2009, стр. 355-360.

[4] Chen, X., Garfinkel, T., Lewis, E. C., Subrahmanyam, P., Waldspurger, C. A., Boneh, D., Dwoskin, J., Ports, D. R. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In Proceedings of the 13th international Conference on Architectural Support For Programming Languages and Operating Systems, ACM, 2008, pp. 2-13.

[5] Adams, K., Agesen, O. A comparison of software and hardware techniques for x86 virtualization. In Proceedings of the 12th international Conference on Architectural Support For Programming Languages and Operating Systems, ACM, 2006, pp. 2-13.

[6] VirtualSquare: Remote System Call. http://wiki.virtualsquare.org/index.php/Remote_System_Call

[7] Sun Microsystems, Inc. RPC: Remote Procedure Call. Protocol Specification. Version 2. Network working group. RFC 1057. 1988.

[8] Sun Microsystems, Inc. NFS: Network File System Protocol Specification. Network working group. RFC 1094. 1989.

[9] Shah. A. Deep Virtue: Kernel-based virtualization with KVM. Linux Magazine (86), 2008, pp. 37-39.