Занятие 4. Ядро Linux. Виртуализация.

4.1. Ядро Linux

4.1.1. Процесс досистемной загрузки. Этапы загрузки.

Загрузка системы состоит из нескольких стадий или этапов. На первом этапе процессор компьютера выполняет код из BIOS. Этот код имеет малый размер и обычно выполняет две основные функции: проверка состояния оборудования после включения (POST – Power On Self Test) и обнаружение загрузочного устройства. В качестве загрузочного может выступать устройство хранения данных: жесткий диск, компакт-диск, USB-флеш, а также внешнее сетевое устройство хранения.
В стандартном случае загрузка производится с локального диска, на котором в MBR (Master Boot Record) содержится код первичного загрузчика (Stage 1) и таблица разделов. MBR имеет размер 512 байт и может быть просмотрена следующими командами:

```
# dd if=/dev/hda of=mbr.bin bs=512 count=1
# od -xa mbr.bin
```

Основная задача первичного загрузчика – запуск вторичного загрузчика (Stage 2). Для этого первичный загрузчик сканирует таблицу разделов в поисках активного раздела. Когда активный раздел обнаружен, производится чтение его загрузочной записи в память и начинается ее выполнение.
Вторичный загрузчик также называется загрузчиком ядра, поскольку основная его задача – это загрузка ядра и RAM-диска с последующей передачей управления ядру операционной системы. Вторичный загрузчик может не только загружать ядро, но и настраивать его. Настройки ядра нужны для временного изменения его функциональности: например, чтобы выбрать другой графический режим виртуальных консолей, чтобы отключить поддержку дополнительных возможностей внешних устройств (если аппаратура их не поддерживает), чтобы передать самому ядру указания, как загружать систему и т.п.
Очень часто конфигурация вторичного загрузчика предусматривает несколько вариантов загрузки, начиная от нескольких вариантов загрузки одного и того же ядра с разными настройками (например, стандартный профиль и профиль с отключенными расширенными возможностями) и заканчивая вариантами загрузки разных ядер и даже разных операционных систем.
Для Линукс на архитектуре x86 существует два основных загрузчика, каждый из которых включает в себя как первичный, так и вторичный загрузчик:

•LILO (Linux Loader) – простой устаревший вариант загрузчика, способен оперировать только дисковыми секторами, соответственно, требует точного указания местоположения ядра и/или RAM-диска с привязкой в географии жесткого диска компьютера.
•GRUB (GRand Unified Bootloader) – универсальный загрузчик следующего поколения. Отличается тем, что включает в себя базовые средства работы с файловыми системами. Эти средства содержатся в так называемом "полуторном" загрузчике (Stage 1.5), для каждой из поддерживаемых файловых систем используется собственный образ загрузчика Stage 1.5. Это позволяет "полуторному" загрузчику производить поиск вторичного загрузчика по имени в файловой системе.

4.1.2. Понятия ядра системы. Загрузка ядра.

Все действия, которые нельзя доверить отдельной подзадаче (процессу) системы, выполняются ядром. Доступом к оперативной памяти, сети, дисковым и прочим внешним устройствам заведует ядро. Ядро запускает и регистрирует процессы, управляет разделением времени между ними. Ядро реализует разграничение прав и вообще определяет политику безопасности, обойти которую, не обращаясь к нему, нельзя просто потому, что в Linux больше никто не предоставляет подобных услуг.
Ядро работает в специальном режиме, так называемом "режиме супервизора", позволяющем ему иметь доступ сразу ко всей оперативной памяти и аппаратной таблице задач. Процессы запускаются в "режиме пользователя": каждый жестко привязан ядром к одной записи таблицы задач, в которой, в числе прочих данных, указано, к какой именно части оперативной памяти этот процесс имеет доступ. Ядро постоянно находится в памяти, выполняя системные вызовы - запросы от процессов на выполнение этих подпрограмм.
Функции ядра после того, как ему передано управление, и до того, как оно начнет работать в штатном режиме, выполняя системные вызовы, сводятся к следующему:

•Сначала ядро определяет аппаратное окружение. Одно и то же ядро может быть успешно загружено и работать на разных компьютерах одинаковой архитектуры, но с разным набором внешних устройств. Задача ядра - определить список внешних устройств, составляющих компьютер, на котором оно оказалось, классифицировать их (определить диски, терминалы, сетевые устройства и т.п.) и, если надо, настроить. При этом на системную консоль (обычно первая виртуальная консоль Linux) выводятся диагностические сообщения (впоследствии их можно просмотреть утилитой dmesg).

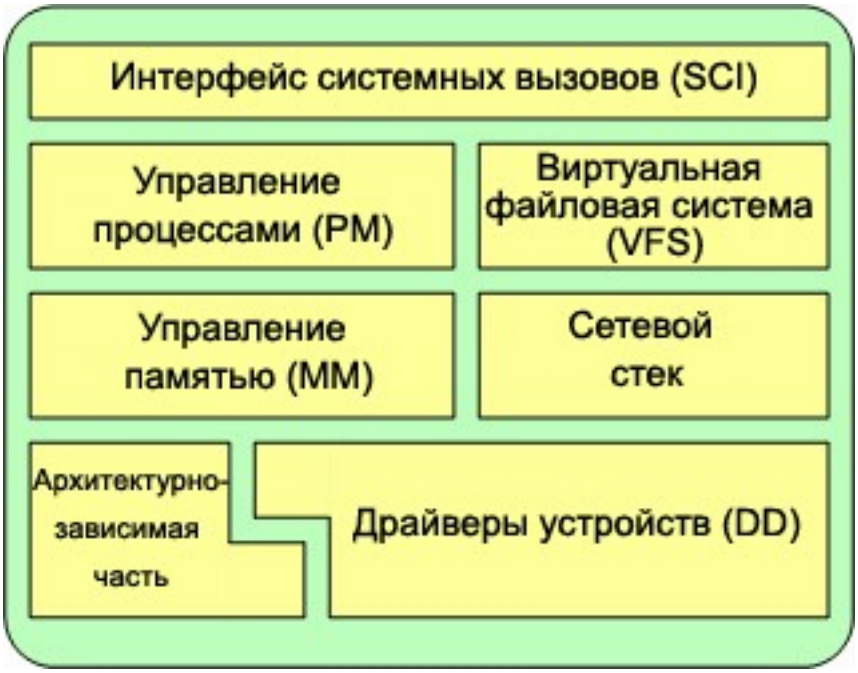
•Затем ядро запускает несколько процессов ядра. Процесс ядра - это часть ядра Linux, зарегистрированная в таблице процессов. Такому процессу можно послать сигнал и вообще пользоваться средствами межпроцессного взаимодействия, на него распространяется политика планировщика задач, однако никакой задаче в режиме пользователя он не соответствует - это просто еще одна ипостась ядра . Команда ps -ef показывает процессы ядра в квадратных скобках, кроме того, в Linux принято (но не обязательно), чтобы имена таких процессов начинались на "k": [kswapd], [keventd] и т.п.

•Далее ядро подключает (монтирует) корневую файловую систему в соответствии с переданными параметрами. Подключение это происходит в режиме "только для чтения" (read-only): если целостность файловой системы нарушена, данный режим позволит запустить утилитуfsck. Позже, в процессе загрузки, корневая файловая система подключится на запись.

•Наконец, ядро запускает из файла /sbin/init первый настоящий процесс. Идентификатор процесса (PID) у него равен единице, он - первый в таблице процессов, даже несмотря на то, что до него там были зарегистрированы процессы ядра. Процесс init - очень старое изобретение, он чуть ли не старше самой истории UNIX, и с давних пор его идентификатор равен 1.

4.1.3. Структура ядра системы. Модули ядра.

В ядре Linux реализован целый ряд важных архитектурных элементов. И на самом общем, и на более детальных уровнях ядро можно подразделить на множество различных подсистем. С другой стороны, Linux можно рассматривать как монолитное целое, поскольку все базовые сервисы собраны в ядре системы.



Интерфейс системных вызовов (SCI)

SCI - это тонкий уровень, предоставляющий средства для вызова функций ядра из пространства пользователя. Как уже говорилось, этот интерфейс может быть архитектурно зависимым, даже в пределах одного процессорного семейства. SCI фактически представляет собой службу мультиплексирования и демультиплексирования вызова функций.

Управление процессами (PM)

Управление процессами сконцентрировано на исполнении процессов. В ядре эти процессы называются потоками (threads); они соответствуют отдельным виртуализованным объектам процессора (код потока, данные, стек, процессорные регистры).

Ядро предоставляет интерфейс программирования приложений (API) через SCI для создания нового процесса (порождения копии, запуска на исполнение, вызова функций Portable Operating System Interface POSIX), остановки процесса (kill, exit), взаимодействия и синхронизации между процессами (сигналы или механизмы POSIX).

Еще одна задача управления процессами - совместное использование процессора активными потоками. В ядре реализован новаторский алгоритм планировщика, время работы которого не зависит от числа

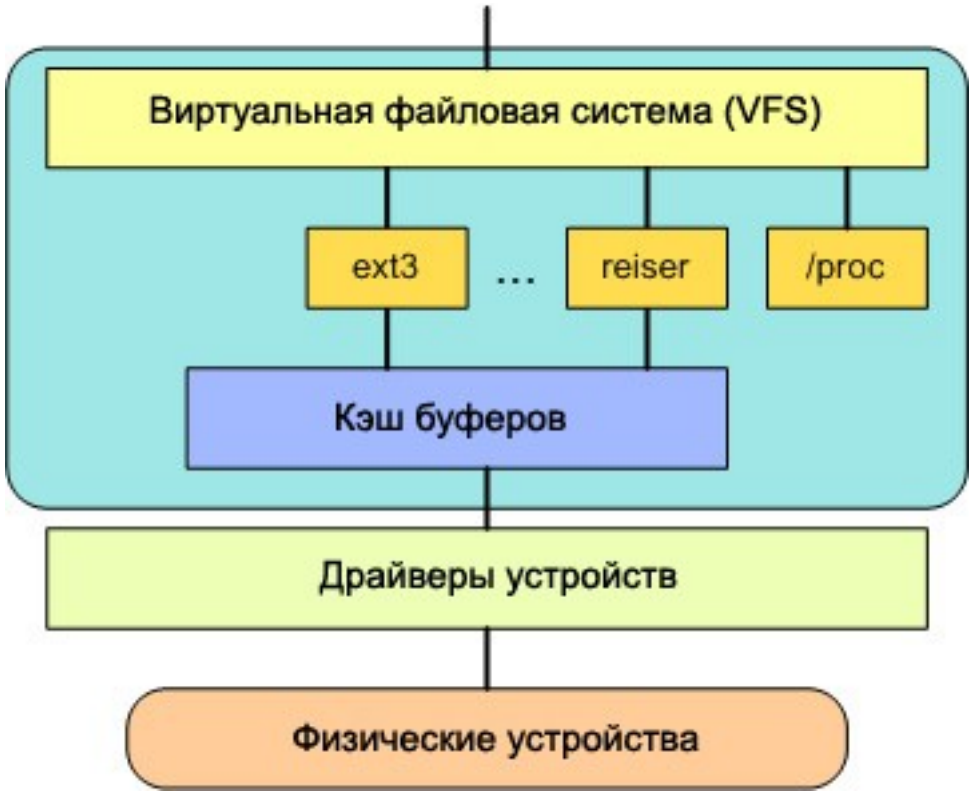
потоков, претендующих на ресурсы процессора. Название этого планировщика - O(1) - подчеркивает, что на диспетчеризацию одного потока затрачивается столько же времени, как и на множество потоков. Планировщик O(1) также поддерживает симметричные многопроцессорные конфигурации (SMP).

Управление памятью (MM)

Другой важный ресурс, которым управляет ядро - это память. Для повышения эффективности, учитывая механизм работы аппаратных средств с виртуальной памятью, память организуется в виде т.н. страниц (в большинстве архитектур размером 4 КБ). Однако управление памятью - это значительно больше, чем просто управление буферами по 4 КБ. Linux предоставляет абстракции над этими 4 КБ буферами, например, механизм распределения slab allocator. Этот механизм управления базируется на 4 КБ буферах, но затем размещает структуры внутри них, следя за тем, какие страницы полны, какие частично заполнены и какие пусты. Это позволяет динамически расширять и сокращать схему в зависимости от потребностей вышележащей системы.
В условиях наличия большого числа пользователей памяти возможны ситуации, когда вся имеющаяся память будет исчерпана. В связи с этим страницы можно удалять из памяти и переносить на диск. Этот процесс обмена страниц между оперативной памятью и жестким диском называется подкачкой(swap).

Виртуальная файловая система

Еще один интересный аспект ядра Linux - виртуальная файловая система (VFS), которая предоставляет общую абстракцию интерфейса к файловым системам. VFS предоставляет уровень коммутации между SCI и файловыми системами, поддерживаемыми ядром.



На верхнем уровне VFS располагается единая API-абстракция таких функций, как открытие, закрытие, чтение и запись файлов. На нижнем уровне VFS находятся абстракции файловых систем, которые определяют, как реализуются функции верхнего уровня. Они представляют собой подключаемые модули для конкретных файловых систем (которых существует более 50).
Ниже уровня файловой системы находится кэш буферов, предоставляющий общий набор функций к уровню файловой системы (независимый от конкретной файловой системы). Этот уровень кэширования оптимизирует доступ к физическим устройствам за счет краткосрочного хранения данных (или упреждающего чтения, обеспечивающего готовность данных к тому моменту, когда они понадобятся).
Ниже кэша буферов находятся драйверы устройств, реализующие интерфейсы для конкретных физических устройств.

Сетевой стек

Сетевой стек по своей конструкции имеет многоуровневую архитектуру, повторяющую структуру самих протоколов. Протокол Internet Protocol (IP) - это базовый протокол сетевого уровня, располагающийся ниже транспортного протокола Transmission Control Protocol, TCP). Выше TCP находится уровень сокетов, вызываемый через SCI.
Уровень сокетов представляет собой стандартный API к сетевой подсистеме. Он предоставляет

пользовательский интерфейс к различным сетевым протоколам.

Драйверы устройств

Подавляющее большинство исходного кода ядра Linux приходится на драйверы устройств, обеспечивающие возможность работы с конкретными аппаратными устройствами. В дереве исходных кодов Linux имеется подкаталог драйверов, в котором, в свою очередь, имеются подкаталоги для различных типов поддерживаемых устройств, таких как Bluetooth, I2C, последовательные порты и т.д.

Архитектурно-зависимый код

Хотя основная часть Linux независима от архитектуры, на которой работает операционная система, в некоторых элементах для обеспечения нормальной работы и повышения эффективности необходимо учитывать архитектуру. В подкаталоге ./linux/arch находится архитектурно-зависимая часть исходного кода ядра, разделенная на ряд подкаталогов, соответствующих конкретным архитектурам. Все эти каталоги в совокупности образуют BSP. В случае обычного настольного ПК используется каталог i386. Подкаталог для каждой архитектуры содержит ряд вложенных подкаталогов, относящихся к конкретным аспектам ядра, таким как загрузка, ядро, управление памятью и т.д. Исходные коды архитектурно-зависимой части находятся в ./linux/arch.

Следует отметить, что ядро Linux является динамическим (поддерживает добавление и удаление программных компонентов без остановки системы). Эти компоненты называются динамически загружаемыми модулями ядра. Их можно вводить в систему при необходимости, как во время загрузки (если найдено конкретное устройство, для которого требуется такой модуль), так и в любое время по желанию пользователя.

Для загрузки модулей в ядро в процессе работы системы используется команда `modprobe`. Команда принимает в качестве аргумента имя модуля или его часть. Поиск модулей производится в каталоге `/lib/modules/`uname -r``. Ключ `-l` выводит пути до файлов модулей, названия которых совпадают или включают указанную строку. Файл `/etc/modprobe.conf` позволяет задать параметры для модулей в случае необходимости.

Узнать, какие модули в настоящее время загружены в ядро, можно с помощью команды `lsmod`. Фактически, эта команда выводит содержимое каталога `/proc/modules` в читаемом формате.

RAM-диск

Возможность динамической загрузки модулей позволяет значительно сократить и упростить код ядра. Однако при этом возникает необходимость сохранить за ядром способность распознавать и работать с большим количеством разнообразных переферийных устройств на этапе загрузки, например, с устройствами хранения данных, с которых производится собственно загрузка ядра, сетевых интерфейсных устройств, и так далее. Для решения этой задачи используется механизм виртуального диска (RAM-диск). Этот диск представляет собой компактный образ файловой системы `tmpfs`, который содержит необходимые модули ядра и сценарии их загрузки. Загрузка RAM-диска производится средствами вторичного системного загрузчика (GRUB или LILO) вместе с загрузкой ядра.

4.1.4. Загрузка системы. Стартовые сценарии. Уровни выполнения.

После загрузки ядра, инициализации оборудования и файловой системы выполняется запуск системных служб. Системные службы как правило запускаются в фоновом режиме, находятся в таблице процессов, однако большую часть времени бездействуют, ожидая запросов. В ранних версиях UNIX все службы, которые запускались при старте системы, записывались в порядке запуска в файле `/etc/inittab`. Из-за недостаточной гибкости такой системы она была вытеснена системой стартовых сценариев. Стартовый сценарий - это программа (обычно на языке shell), которая описывает и реализует логику запуска и останова какой-либо системной службы (или группы служб). Обычно параметры, передаваемые стартовым сценариям, единообразны в различных дистрибутивах Линукс. Как правило это командное слово `start`, `stop`,`status`. Некоторые службы поддерживают расширенный формат команд для стартовых сценариев, например, для `httpd` поддерживаются также команды `restart`, `reload`, `fullstatus`, `configtest` и другие.

Все стартовые сценарии служб обычно хранятся в каталоге `/etc/init.d` или `/etc/rc.d/init.d`. Остановить или запустить службу можно, вызвав соответствующий сценарий с командой. Во многих дистрибутивах для управления службами можно использовать команду `service`.

Чаще всего при загрузке системы приходится выполнять не все сценарии, размещенные в `/etc/init.d`, а только какую-то часть. При этом в некоторых случаях сценарии могут выполняться с различными командами. Для этого в ранних версиях UNIX использовался сценарий`/etc/rc`, в который заносились все нужные для запуска команды. При этом изменение порядка и состава загрузки служб требовало редактирования файла.

Более современным подходом является использование "схемы .d". В рамках этой схемы для запуска стартовых сценариев создается отдельный каталог `/etc/rc.d`, в котором создаются ссылки только на те сценарии из `/etc/init.d`, которые необходимо запустить. Запуском занимается по-прежнему общий стартовый сценарий `/etc/rc`. Ссылки оформляются так, чтобы можно было определить порядок запуска и команду, которую нужно передать сценарию. Для этого в начале ссылки ставится буква S (start) или K (kill). Затем указывается двухзначное число, определяющее порядковый номер

сценария при запуске.
Стартовые сценарии могут быть сгруппированы для создания различных вариантов загрузки системы и системных служб.

Уровни выполнения.

В Линукс предусмотрено несколько вариантов начальной загрузки системы, называемых уровнями выполнения (run levels):
0. Остановка системы. Все службы остановлены, диски размонтированы. Электропитание отключается программно, если это поддерживается оборудованием.
1. Однопользовательский режим загрузки системы. В системе не запускается никаких служб. Доступна единственная системная консоль с доступом уровня суперпользователя.
2. Многопользовательский режим с отключенной сетью. Сетевые службы не запускаются в автоматическом режиме, но могут быть запущены и настроены вручную.
3. Многопользовательский сетевой режим. Запущены все службы, включая сетевые. Этот уровень обычно устанавливается стандартным для серверных систем.
4. Зарезервирован.
5. Многопользовательский графический режим. На этом уровне обычно функционируют рабочие станции, предоставляя пользователям возможность работать с графической подсистемой X11. Могут не запускаться некоторые сетевые службы.
6. Перезагрузка системы. Все службы остановлены, диски размонтированы. Инициируется перезагрузка системы, если поддерживается оборудованием.
7-9. Зарезервированы.
Зарезервированные уровни загрузки могут использоваться администратором для того, чтобы определить особые профили работы системы. Переход с уровня на уровень осуществляется командой init N, где N – номер уровня.
Конфигурация уровня выполнения по умолчанию находится в файле /etc/inittab. Выяснить текущий уровень выполнения позволяет команда runlevel.
Обычно каждому уровню выполнения соответствует собственный каталог /etc/rcN.d/ или /etc/rc.d/rcN.d/. При переходе между уровнями сначала в указанном порядке запускаются с командой stop все сценарии, имена которых начинаются на K. Затем, также по порядку, запускаются с командой start сценарии, имена которых начинаются на S.
Для автоматического управления уровнями выполнения используется команда chkconfig. Она позволяет просмотреть текущие настройки служб по уровням выполнения с помощью ключа --list:

```
root@localhost root]# chkconfig --list netfs
  netfs      0:off  1:off  2:off  3:on  4:on  5:on  6:off
```

Также с помощью chkconfig можно включить или отключить службу на определенном уровне:

```
root@localhost root]# chkconfig --level 0,6 ntpd off
```

4.2. Виртуализация

4.2.1. Понятие виртуализации.

Существует несколько типов виртуализации, работающие на различных уровнях абстракции. Два основных метода виртуализации в Linux – это полная виртуализация и паравиртуализация. Менее распространены методы эмуляции оборудования и виртуализации операционной системы.

Эмуляция оборудования

Сложный вид виртуализации, при которой одиночный компьютер может представляться как множественная архитектура. При этом моделирование команд происходит на уровне специализированного оборудования. Недостатком этих систем является пониженное быстродействие. Примеры систем эмуляции оборудования: `Bochs`, `QEMU`.

Виртуализация ОС

Наиболее простая форма виртуализации, в результате которой одиночный компьютер работает с несколькими операционными системами одного типа. Этот тип виртуализации просто изолирует некоторые приложения на отдельной операционной системе (что означает, что все должны использовать один и тот же тип и версию операционной системы).
Примером такой системы виртуализации являются LXC (Linux Containers), `jail`, `openvz`. Основное преимущество такой системы – повышенное быстродействие. Проблемы могут возникать с разделением прав доступа к оборудованию и прочими средствами обеспечения безопасности.

Полная виртуализация

Эта модель использует виртуальную машину, которая осуществляет связь между гостевой операционной системой и аппаратными средствами хост-системы. Посредничество между системой и оборудованием осуществляет гипервизор. Внутри него функционируют механизмы защиты и изоляции, поскольку аппаратные средства требуется разделять между несколькими гостевыми системами.

Примеры полной виртуализации – KMV (Kernel virtual machine), VMware.

Паравиртуализация

Этот метод использует гипервизор для разделения доступа к основным аппаратным средствам, но объединяет код, касающийся виртуализации, в непосредственно операционную систему. При этом гостевая ОС кооперируется с гипервизором, что устраняет потребность в перекомпиляции и перехватывании команд. Очевидно, что для этого необходимо внести изменения в ядро гостевой ОС.

Пример систем паравиртуализации – Xen.

Преимущества паравиртуализации – повышение производительности виртуальных машин почти до значений, почти не отличающихся от реальных систем.

4.2.2. Средства виртуализации в Linux

Еще одно усовершенствование Linux - возможность ее использования в качестве операционной системы для других операционных систем (т.н. гипервизора). В ядро было внесено усовершенствование, получившее название Kernel-based Virtual Machine (KVM, виртуальная машина на базе ядра). В результате этой модификации в пространстве пользователя был реализован новый интерфейс, позволяющий исполнять поверх ядра с поддержкой KVM другие операционные системы. В таком режиме можно не только исполнять другие экземпляры Linux, но и виртуализовать Microsoft® Windows®. Единственное ограничение состоит в том, что используемый процессор должен поддерживать новые инструкции виртуализации.

Подход, который использует KVM, состоит в том, чтобы превратить ядро Linux в гипервизор простой загрузкой модуля ядра. Модуль ядра экспортирует устройство, называемое `/dev/kvm`, которое делает возможным гостевой режим ядра (вдобавок к обычным режимам ядра и пользователей). С `/dev/kvm` VM имеет свое собственное адресное пространство, отдельное от адресного пространства ядра или любых других работающих VM. `/dev/kvm` отличается тем, что каждый процесс, который его открывает, видит другую карту (для поддержания изоляции виртуальных машин).

Модуль KVM вводит в ядро новый способ исполнения. В то время как обычное ядро поддерживают режим ядра и режим пользователя, KVM представляет новый гостевой режим (guest). Гостевой режим используется, чтобы выполнить все команды, не связанные с операциями ввода/вывода (I/O), в то время как нормальный пользовательский режим осуществляет ввод/вывод для гостевой ОС.

Использование KVM

Для того, чтобы создать образ диска для виртуальной машины, можно использовать команду qemu-img:

```
$ qemu-img create -f qcow vm-disk.img 4G
```

После создания виртуального диска можно загрузить на него образ гостевой ОС:

```
$ kvm -no-acpi -m 384 -cdrom guestos.iso -hda vm-disk.img -boot d
```

Использование virt-install.

Для создания виртуальной машины, совмещенного с установкой на нее гостевой операционной системы, можно воспользоваться пакетом `python-virtinst`. Этот пакет устанавливает программу `virt-install`, которая использует библиотеку `libvirt` для подключения к гипервизору KVM и конфигурирования виртуальных машин. При этом можно указать любой источник для установки, включая сетевое расположение, например URL или каталог NFS.

Программа virt-manager - GUI для virt-install.