ИУ-10

Системное

Программное

Обеспечение

Системы виртуализации Контейнеризация (виртуализация на уровне ядра ОС)

Москва, 2022

На этом уроке

- Рассмотрим принципы создания изолированных окружений (контейнеров).
- Изучим особенности использования контейнеров и их отличия от полноценных виртуальных машин.
- Познакомимся с примерами реализации: Virtuozzo и OpenVZ, Docker.

Содержание

Принципы создания изолированных окружений

На вводном занятии, посвящённом обзору существующих сегодня систем виртуализации, мы кратко затронули тему легковесной виртуализации, или, как сейчас модно говорить, «контейнеров». Суть легковесной виртуализации заключается в запуске нескольких пользовательских пространств поверх одного ядра операционной системы. Пользовательское пространство (userland или user space в английской терминологии) — это весь набор программного обеспечения, выполняемого поверх ядра операционной системы.

С точки зрения ядра ОС всё ПО, работающее в пространстве пользователя, отличается друг от друга лишь уникальными идентификаторами и расположенными в разных частях памяти управляющими структурами. Как эти программы и библиотеки взаимодействуют друг с другом, ядру большого дела нет. Однако для пользовательского ПО, как это ни странно, важно то окружение, в котором оно находится.

Современное ПО из-за своей высокой сложности часто полагается на использование внешних библиотек — наборов подпрограмм или объектов, используемых разными программами. Это могут быть:

- динамические библиотеки (файлы с расширением .dll в Microsoft Windows, .so в Linux дистрибутивах, .dylib в macOS);
- библиотеки для специфических языков, таких как Python, JavaScript и т. д.

Таких зависимостей множество: непосредственных, прямых или непрямых — по типу компонентов, от которых зависит конкретный компонент. А сами компоненты, от которых зависят другие, могут иметь несколько версий. Часто приходится говорить о десятках версий, обусловленных разработкой новых версий библиотек, а также разнообразными их конфигурациями. Можно запросто оказаться в ситуации, когда целевая программа может быть запущена только в каком-то конкретном окружении. Эта проблема, в частности, решается созданием дистрибутива — набора компонентов, заведомо сочетающихся друг с другом. В нём все зависимости удовлетворены.

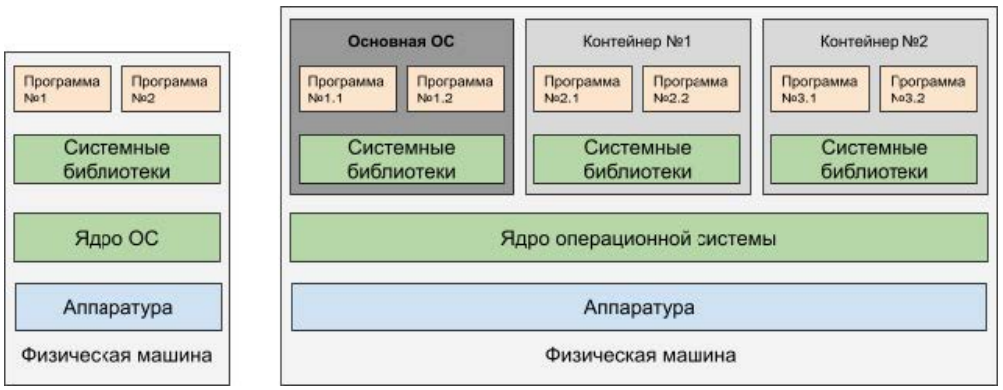
Дистрибутив (англ. distribute — «распространять») — это форма распространения программного обеспечения.

С другой стороны, некоторым программам важно, какими характеристиками они обладают в данном окружении. В частности, системы инициализации, такие как systemd, предполагают, что они должны быть запущены первым процессом в системе, ведь они должны организовать запуск

всех остальных компонентов пользовательского пространства. В противном случае их запуск не имеет никакого смысла и они завершают свою работу, так ничего и не сделав. Более того, тот же демон инициализации предполагает, что файлы, необходимые ему для работы, а также программы, которые нужно будет стартовать, находятся по вполне конкретным путям на файловой системе. Причём совпадающим с путями уже работающего окружения.

То есть запустить ещё один экземпляр дистрибутива ОС в окружении другого, уже работающего, дистрибутива просто так не получится. Как минимум для второго дистрибутива необходимо создать окружение, в котором он будет наблюдать такое окружение, как если бы он единственный был запущен поверх ядра ОС. Соответственно, если такие условия создать, то можно будет говорить о запуске нескольких экземпляров операционных систем, что уже начинает подходить под определение системы виртуализации — запуск ПО в окружении, очень похожем на некую отдельную физическую машину.

В разговоре о контейнерах термин «операционная система» приобретает особый смысл. Он описывает фактически только часть полноценной ОС, которая работает в пространстве пользователя, то есть ядро ОС в данном случае выносится за скобки. Единственное исключение — это хозяйская ОС, частью которой является одно и единственное ядро, предоставляющее сервисы в том числе и контейнеризованным ОС.



Посмотрим, что же нам нужно для создания «виртуальной машины», то есть такого окружения, при котором ПО (операционная система) будет функционировать как будто на отдельном физическом компьютере. Можно сформулировать два базовых требования:

1. **Изолированная файловая система**. Таким образом ОС внутри контейнера сможет распоряжаться всем доступным ей накопителем данных, не видя файлов хозяйской ОС. Так она не будет представлять опасность для хозяйской системы, файлы которой останутся в сохранности независимо от того, что будет делать ОС в контейнере. Это в

значительной мере правдиво до тех пор, пока для удобства решения некоторых задач в контейнер не оказывается подключена файловая система хозяйской ОС. Вот тогда появляется более чем достаточно возможностей навредить хозяйской системе.

2. **Изолированные от хозяйской системы процессы**. Изолированные в том смысле, что процессы, запущенные в контейнере, не должны знать о существовании процессов хозяйской системы. Это нужно для решения одновременно двух задач: предотвращения влияния ОС в контейнере на хозяйскую систему и обеспечения привычного окружения для ПО в контейнере. Иначе непонятно, как демону инициализации в контейнере поступать с уже запущенным таким же демоном хозяйской системы.

Интересно заметить, что изоляция памяти, доступной контейнеру, не фигурирует в требованиях: ни в базовых, ни в дополнительных. По определению контейнер содержит только процессы в пользовательском пространстве, которые, как мы знаем, и так работают каждый в своём адресном пространстве, а всё управление памятью возложено на ядро ОС.

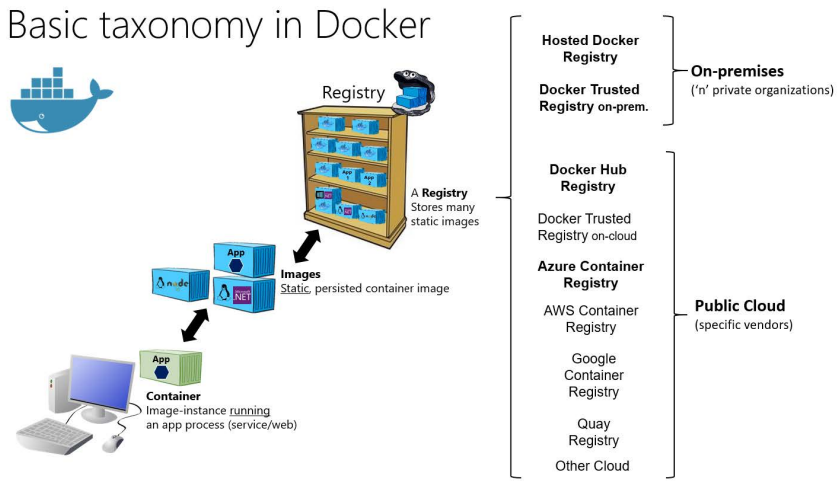
Тем не менее для более качественной изоляции или, если угодно, виртуализации, стоит задуматься над следующими требованиями:

1. **Ограничение доступа ПО из контейнера к аппаратуре**. Поскольку ПО в контейнере не знает о существовании другой, параллельно работающей, системы, оно может попробовать получить доступ к какому-то реальному оборудованию и монопольно использовать его. Скорее всего, это не совсем то, чего бы нам хотелось. Не исключены и такие ситуации, когда какому-то контейнеру умышленно выделяется какой-то аппаратный ресурс для монопольного использования.

2. **Управление ресурсами, которые доступны ПО в контейнере**. Самое простое — это объём доступного процессорного времени и оперативной памяти. В противном случае ПО контейнера сможет захватить всю доступную вычислительную мощность и память хозяйской системы, тем самым нарушив её нормальную работу. Более сложные случаи — это ограничение используемой пропускной способности сетевого соединения, интерфейсов ввода-вывода, доступного объёма накопителя.

Немного терминологии

Разобравшись с общими принципами, лежащими в основе современных систем контейнеризации, стоит познакомиться с терминологией. В противном случае легко запутаться в этой непростой, но очень популярной теме.



Basic taxonomy in Docker, Контейнеры, образы и реестры Docker

Контейнер (Container)

Запущенный процесс или группу процессов в изолированном окружении принято называть контейнером. Каждый такой контейнер — это уникальный объект. Он имеет свой идентификатор в системе, и именно в нём происходит настоящая полезная работа. Отсюда и такое частое употребление этого термина.

Образ контейнера (Container image)

Если же контейнер остановить, то всё, что от него останется, — это некоторый набор файлов на файловой системе. И вот этот набор файлов, описывающих состояние остановленного контейнера, называется образом контейнера или просто образом. Причём в зависимости от используемой системы управления контейнерами набор файлов и их особенности могут различаться — это задаётся форматом образа контейнера.

Сейчас OCI-образ — это наиболее популярный формат. В частности, актуальные версии Docker используют именно этот формат образов. Что интересно, формат OCI-образа не зависит ни от архитектуры процессора хозяйской системы, ни от используемой операционной системы в том смысле, что этот формат может описать полное состояние контейнера на любой системе. Однако, разумеется, один и тот же образ невозможно использовать на процессорах разных архитектур или в хозяйских системах с разными ОС. Нюанс может заключаться в том, что контейнеры могут запускаться внутри виртуальной машины и тем самым может создаваться видимость запуска Linux-контейнера в Windows и наоборот.

Базовый образ (Base image)

Как правило, контейнер создаётся не с нуля (это возможно, но используется редко), а из заранее подготовленного образа. Причём этот образ, в сущности, — зафиксированное состояние ранее использованного контейнера. То есть такие образы можно создавать, интерактивно делая изменения в текущем контейнере. Однако чаще образы создают при помощи специального инструментария: Docker и Dockerfile, Buildah и т. д. Такие образы называют базовыми, так как используют их как базу для своего продукта. Самый простой пример — это минимальная файловая система какого-нибудь популярного Linux-дистрибутива, например Ubuntu, Alpine и т. д.

Реестр (Registry)

Подготовленные образы можно хранить и использовать локально, но хорошая практика состоит в публикации образов в соответствующих реестрах. Реестр образов — это некое подобие библиотеки, в которой хранятся образы. Причём реестры используются не только и не столько для хранения образов. Имея информацию о содержимом образа, а ещё лучше — использованный для его создания Makefile, Dockerfile или shell-скрипт, можно запросто воссоздать образ самостоятельно. Реестры нужны также для обмена образами и их распространения.

Наиболее известный на сегодняшний день реестр — это Docker Hub. Там можно найти образы большинства популярных Linux-дистрибутивов, а также массу производных от них. Более того, любой желающий может опубликовать самостоятельно подготовленный образ. К сожалению, найти актуальную статистику по использованию Docker Hub весьма непросто, однако можно обратиться к обзору из статьи на Habr.com.

Существует и масса других реестров, начиная с публичных (Fedora Container Registry, Red Hat Quay) и заканчивая корпоративными или личными.

Среда исполнения контейнеров (Container runtime)

Выше мы определили контейнер как набор процессов, запущенных в изоляции от хозяйской системы. И образ контейнера как набор файлов, описывающих статическое состояние контейнера, когда он остановлен. Соответственно, для воссоздания работающего контейнера из образа необходимо:

- в соответствии с форматом образа прочитать из него информацию о конфигурации изолированной среды для запуска контейнера;

- прочитать файловую систему контейнера;
- создать и запустить необходимые процессы внутри изолированного окружения.

Перечисленный набор операций выполняет низкоуровневая среда исполнения контейнеров. Container runtime — это некая аналогия среды выполнения вроде Java Runtime Environment (JRE), необходимая для запуска байт-кода Java. Разница в том, что container runtime только создаёт необходимое окружение и управляет жизненным циклом контейнера, а приложения или скрипты внутри контейнера исполняются точно так же, как и приложения хозяйской системы без использования runtime-окружения. Причём одной только среды исполнения контейнеров вполне достаточно для создания и использования контейнеров. Типичный пример — runC.

runC — утилита командной строки, предназначенная для создания и запуска контейнеров в соответствии со спецификацией Open Container Initiative (OCI).

Помимо runC, написанного почти полностью на Go, существует crun. Это тоже Container Runtime, реализующий спецификации OCI, но написанный на чистом Си. По сравнению с runC имеет более высокую производительность и использует меньший объём памяти хозяйской системы при работе.

Container engine

Для упрощения работы с контейнерами функционал container runtime зачастую существенно расширяется добавлением API удалённого управления, утилитами работы с образами контейнеров и даже средствами удобного обмена образами и контейнерами. Такие комбайны принято называть высокоуровневыми средствами для работы с контейнерами или container engine. В качестве примера можно привести containerd или podman.

В какую же категорию отнести Docker, мы узнаем позже, когда будем говорить именно о нём, а пока просто примем к сведению, что Docker — это нечто большее.

Системный контейнер

Первоначально идея виртуального окружения, впоследствии названного контейнером, сводилась к запуску нескольких копий пользовательских пространств полноценной ОС на одной машине. Такие контейнеры с полноценной ОС внутри принято называть системным контейнером или контей-

нером операционной системы (ОС). Какое-то время системные контейнеры составляли серьёзную конкуренцию полноценным виртуальным машинам, особенно на рынке веб-хостинга.

Контейнер приложения

Постепенно пришло понимание, что, имея уже одну полноценную ОС (имеется в виду хозяйская ОС), нет необходимости запускать её копию со всеми десятками или сотнями служебных процессов, если фактически необходимо запустить всего лишь одно приложение, например веб-сервер. Так появились контейнеры приложений, которые содержат всё необходимое для запуска одного или нескольких приложений или сервисов, а не полноценной ОС. Они занимают меньше места на диске и в памяти, потребляют меньше ресурсов процессора хозяйской системы и позволяют реализовать ставшую сейчас чрезвычайно популярной микросервисную архитектуру.

Микросервисная архитектура — вариант сервис-ориентированной архитектуры программного обеспечения, ориентированный на взаимодействие насколько это возможно небольших, слабо связанных и легко изменяемых модулей — микросервисов. Получил распространение в середине 2010-х годов в связи с развитием практик гибкой разработки и DevOps.

Может даже оказаться, что контейнеры приложений гораздо более популярны, чем их системные предки.

В Linux

Изоляция и управление группой процессов в Linux реализуются благодаря cgroups и namespaces.

Пространство имён (англ. namespaces) — это функция ядра Linux, позволяющая изолировать и виртуализировать глобальные системные ресурсы множества процессов.

При помощи namespaces происходит виртуализация или изоляция самых разнообразных ресурсов. А cgroups позволяют эти виртуализированные ресурсы делать доступными только некоторым процессам или группам (иерархиям или деревьям) процессов, а кроме того, управлять объёмом доступных ресурсов. Точнее, вводить ограничения на их использование. И современные инструменты контейнеризации в Linux построены с использованием упомянутых выше двух механизмов ядра.

Если сильно упростить, Docker, LXC, lmctfy или runC — это набор скриптов, которые самостоятельно настраивают namespaces и cgroups, а

после этого запускают указанное приложение в только что созданном контейнере.

Cgroups

Контрольная группа (англ. control group, cgroups, cgroup) — группа процессов в Linux, для которой механизмами ядра наложена изоляция и установлены ограничения на некоторые вычислительные ресурсы: процессорные, сетевые, ресурсы памяти, ресурсы ввода-вывода. Механизм позволяет образовывать иерархические группы процессов с заданными ресурсными свойствами и обеспечивает программное управление ими. Следующие типы ресурсов хозяйской системы могут быть использованы с cgroups:

1. **CPU** — управляет распределением вычислительных ресурсов центрального процессора. Для каждой группы можно выделить часть процессорного времени ЦП, причём можно указывать либо относительное значение, например 30длительность временного интервала, доступного для работы группы. Можно указать даже приоритет конкретной группы относительно других групп.

2. **Memory** — управляет распределением памяти, позволяя процессам в данной группе использовать ограниченный объём памяти. Если процессы группы попытаются превысить максимально допустимое значение, они будут уничтожены при помощи механизма OOM killer. Это механизм в ядре ОС Linux, позволяющий в случае критического недостатка памяти для работы системы освободить место в ОЗУ за счёт уничтожения некоторых работающих процессов.

3. **IO** — управляет распределением ресурсов подсистем ввода-вывода. Возможно ограничение либо по объёму прочитанных или записанных данных за единицу времени, либо по количеству операций ввода-вывода (IOPS) также за единицу времени.

   IOPS (аббревиатура от англ. input/output operations per second — «количество операций ввода-вывода в секунду»; произносится как «айопс») — количество операций ввода-вывода, выполняемых системой хранения данных за одну секунду. Один из параметров, используемых для сравнения систем хранения данных: жёстких дисков (НЖМД), твердотельных накопителей (SSD), сетевых хранилищ SAN, NAS) и оценки их производительности.

4. **PID** — управляет возможностью процессов группы создавать новые (дочерние) процессы.

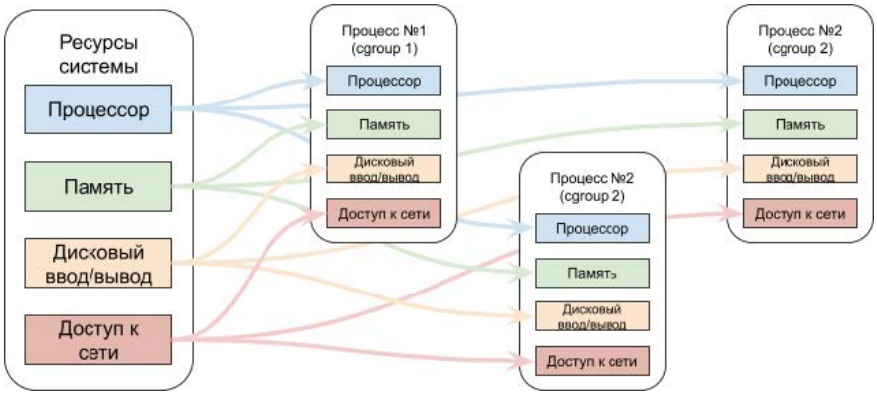
5. **Device controller** — управляет доступом процессов из группы к файлам устройств, обычно расположенным в папке /dev/. Возможно задание ограничений не только на создание новых файлов устройств, но также на доступ к уже имеющимся.

6. **RDMA** — управляет доступом к RDMA-ресурсам.

   Удалённый прямой доступ к памяти (англ. remote direct memory access, RDMA) — аппаратное решение для обеспечения прямого доступа к оперативной памяти другого компьютера при помощи высокоскоростной сети. Такой доступ позволяет получить доступ к данным, хранящимся в удалённой системе, без привлечения средств операционных систем обоих компьютеров. Это метод пересылки данных с высокой пропускной способностью и низкой задержкой сигнала, который особенно полезен в больших параллельных вычислительных системах — кластерах.

7. **Perf_events** — управляет доступом к информации, предоставляемой подсистемой Perf ядра Linux.

   Perf — средство оценки производительности разных систем ядра ОС, приложений пользователя, аппаратуры и т. д. В основе этого инструмента лежит сбор разнообразной статистической информации с возможным последующим её анализом.



Cgroups: управление ресурсами системы, предоставляемыми процессам пользователя

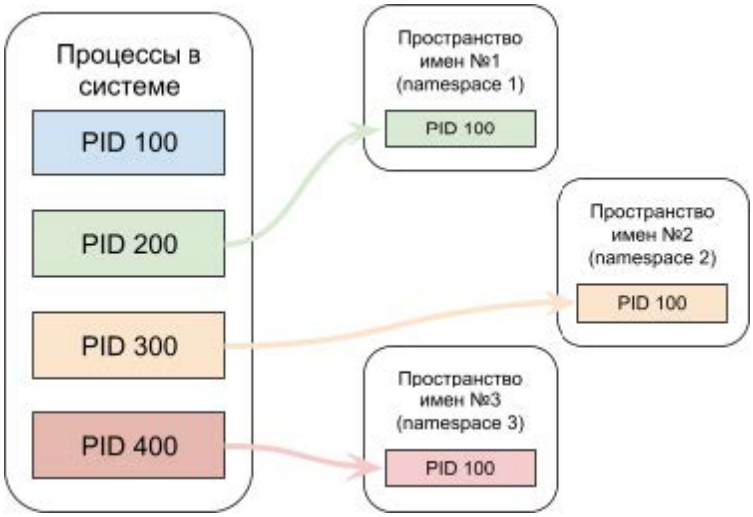
Namespaces

Пространства имён (namespaces) — это функциональная особенность ядра ОС Linux, благодаря которой разные процессы в пространстве пользователя видят разные ресурсы так, как если бы они были одними и теми же.

То есть фактически ОС подменяет реальные свойства своих ресурсов при предоставлении их пользовательским приложениям.

Например, мы знаем, что каждый процесс, запущенный поверх ядра Linux, должен иметь свой уникальный идентификатор — Process ID (PID), назначенный ядром при создании данного процесса. Благодаря namespaces можно сделать так, чтобы процесс считал, что его PID, например, равен 1, что может быть только у первого пользовательского процесса в системе. В большинстве случаев это процесс инициализации, то есть сегодня это обычно systemd.

Более того, дочерние процессы также будут видеть PID своего родителя, равный 1. При этом, разумеется, реальный PID этого процесса, известный только ядру ОС и используемый ядром для управления процессом, будет значительно отличаться от 1. Например, он будет 12345.



PID namespace

На сегодняшний день реализованы такие виды пространства имён:

1. **Файловая система** (Mount, mnt) — позволяет предоставлять каждому процессу или группе процессов свой собственный экземпляр дерева файловой системы. Прообразом этого пространства имён послужила chroot — операция изменения корневого каталога в Unix-подобных операционных системах.

2. **ID процессов (PID)** — позволяет назначать ID процессов независимо в каждом пространстве имён. Таким образом, в разных пространствах имён могут быть процессы с одинаковыми в рамках этих пространств ID.

3. **Сети (Network)** — позволяет виртуализировать сетевой стек. То есть каждое пространство имён будет иметь свой пул IP-адресов и портов, свои таблицы маршрутизации, файервол и так далее.

4. **Межпроцессное взаимодействие (IPC)** — изолирует средства межпроцессного взаимодействия пространства имён, такие как совместно используемая память, очереди сообщений, семафоры. Процессы из разных пространств не будут иметь возможности умышленно или неумышленно взаимодействовать друг с другом.

5. **Пользовательские ID (User)**. Так же, как ID процессов, можно виртуализовать ID пользователей. User ID namespace позволяет сделать именно это: иметь одинаковые UID в разных пространствах имён.

6. **Control group (cgroup)** — виртуализует cgroups.

7. **UTS** — виртуализует имя хоста (hostname)

В Windows

Изначально контейнеры в том виде, в каком мы их знаем сейчас, появились в UNIX-подобных ОС (BSD, Linux). Но благодаря своей высокой популярности они нашли дорогу и в системы с ОС Windows. Доступна документация, описывающая различные варианты реализации и использования контейнеров в Windows.

Первоначально реализация контейнеров происходила путём запуска виртуальной машины с обыкновенным или специализированным Linux-дистрибутивом. Позже компания Microsoft реализовала решение, очень похожее на таковое в BSD и Linux, — изоляцию процессов без запуска виртуальной машины. Кроме того, помимо похожей на BSD и Linux изоляции процессов средствами ядра ОС, была реализована изоляция процессов средствами гипервизора Hyper-V. Причём, несмотря на то, что используется гипервизор Hyper-V, под его управлением запускается не полноценная и независимая копия некой случайной ОС в том смысле, что это может быть любая ОС на выбор пользователя, а клон уже работающей на хозяйской системе ОС. При этом переиспользуются её части: файловая система, установленные приложения и прочее.

Имеется в виду переиспользование только тех частей ПО, которые доступны только для чтения. В основном это исполняемый код, константы и строки. Если бы этот же набор ПО был заново установлен в гостевую ОС, эти части ПО, доступные только для чтения, были бы точно такими же. Изменяемые же части ПО, такие как данные пользователя, разумеется, не должны использоваться совместно гостевой и хозяйской системами.

Таким образом, контейнерная революция добралась и до Windows. Однако стоит помнить о том, что в отличие от виртуальных машин, как правило, не зависящих от хозяйской ОС или гипервизора, контейнер полага-

ется на базовые сервисы хозяйской ОС. А потому Linux-контейнер принципиально невозможно запустить непосредственно на Windows и наоборот. Это становится возможным только в случае использования виртуальной машины с запущенной в ней ОС необходимого типа.

В статье Linux containers on Windows 10 подробно рассматриваются доступные варианты запуска Linux-контейнеров на Windows.

По сравнению с Linux-контейнерами их версия для Windows обладает определёнными недостатками:

1. Не любая версия Windows подходит для запуска контейнеров. Подойдут, например, Windows Server 2016 и Windows 10 Professional или Enterprise, начиная с Anniversary update (версия 1607).

2. Наличие ограничений, связанных с лицензированием. Не на любой системе с Windows можно запустить неограниченное количество Windows-контейнеров.

3. Гораздо меньшее количество реестров с образами Windows-контейнеров.

В macOS

Ситуация с контейнерами в macOS примерно такая же, как в Windows. Разница лишь в том, что своей собственной реализации окружения для запуска контейнеров непосредственно на macOS до сих пор не было разработано. То есть использование контейнеров в macOS ограничивается запуском их в виртуальной машине. До недавних пор всё сводилось к установке VMware Fusion или Oracle VirtualBox, в них устанавливался полноценный Linux-дистрибутив. Далее задача становилась тривиальной — запуск Linux-контейнеров в Linux. Позже в macOS был реализован собственный фреймворк для создания легковесных гипервизоров — Hypervisor.framework, ставший доступным, начиная с macOS 10.10 Yosemite. Появилась возможность отказаться от полноценного внешнего гипервизора и создать специализированный, нацеленный на обслуживание нужд контейнеризованных окружений. Что и сделала компания Docker Inc., создав специально для macOS инструмент под названием HyperKit.

Сравнение контейнеров и виртуальных машин

Производительность

Говоря о производительности, мы подразумеваем скорее обратное — накладные расходы на обслуживание всей активности, не связанной напрямую с выполнением полезной работы гостевой системой. И в этом смысле контейнеры имеют преимущества:

1. Нет необходимости переключаться между режимами гипервизора и режимом работы гостевых систем. Это, как мы помним, достаточно накладная операция.

2. Нет необходимости эмулировать аппаратуру, включая части центрального процессора, такие как MMU.

3. Собственно сам гипервизор отсутствует в системе, а потому на его сервисные функции не приходится тратить процессорное время. Мы избавлены от ещё одного уровня абстракции, который, в отличие от контейнеров, очень часто оказывается реально исполняемым аппаратурой.

Другими словами, контейнеры в общем случае позволяют более эффективно использовать вычислительные ресурсы хозяйской системы. Более высокая эффективность контейнеров проявляется как минимум в двух важных аспектах:

1. **Гораздо более быстрый старт полезной задачи**. Это особенно справедливо для контейнеров приложений, в которых после создания всего необходимого окружения необходимо запустить только целевое приложение, например веб-сервер. Контейнеры операционных систем, содержащие полный набор библиотек и приложений пользовательского пространства, стартуют медленнее, поскольку происходит запуск множества служб и сервисных процессов. Но даже при этом не происходит перезапуска ядра ОС и связанной с этим процессом инициализации аппаратуры, что само по себе может привносить существенные задержки.

2. **Более высокая плотность размещения виртуализованных окружений на одной хозяйской машине**. Очевидно, что для запуска N копий веб-сервера в контейнерах приложений потребуется меньше ресурсов хозяйской системы, нежели для запуска N копий полноценных виртуальных машин с полными ОС и одним веб-сервером в каждой.

Несмотря на высокую скорость работы ЦП, измеряемую в гигагерцах, и высокоскоростные шины данных, такие как PCIe, при инициализации некоторого оборудования происходят задержки. Этому может быть несколько причин. Зачастую внешнее оборудование имеет внутри свой процессор, исполняющий код его же прошивки. Часто эта прошивка загружается во внешнее устройство с хозяйской системы, причём иногда при помощи относительно низкоскоростного интерфейса типа SPI. После того как прошивка была загружена, контроллер внешнего устройства начинает свою процедуру инициализации. Только проверив все внутренние состояния, выполнив необходимые калибровки и включив все свои устройства, он готов к взаимодействию с хозяйской системой.

Ещё хуже дело обстоит с инициализацией дисковых и сетевых устройств. Приводы с вращающимися дисками должны раскрутить эти диски до рабочей скорости в тысячи оборотов в минуту, обычно до 5 400 или 7 200 об/мин. Разумеется, это не происходит мгновенно. А в случае сетевых устройств согласование режима работы на физическом уровне, известное как autonegotiation, может занимать буквально секунды.

Универсальность

В плане универсальности контейнеры проигрывают полноценным виртуальным машинам, так как контейнеры по определению используют ядро хозяйской ОС. Таким образом, невозможно запустить контейнер, предназначенный для работы в Windows, Solaris или FreeBSD, на машине с Linux-дистрибутивом. Современные же гипервизоры могут запускать гостевые системы с разнообразными ОС: Windows, Linux, Solaris и т. д. Причём все эти гостевые системы отлично уживается друг с другом.

Но даже при использовании гостевых систем одного типа, например Linux-дистрибутивов, в системе контейнеризации принципиально невозможно иметь ядро ОС с разной функциональностью. Безусловно, при помощи cgroups в Linux можно ограничить доступ процессов контейнера к имеющейся функциональности хозяйского ядра ОС. Однако добавить дополнительную функциональность ядра ОС для данного контейнера возможно, лишь изменив ядро хозяйской системы. Что в свою очередь может потребовать перезапуска ядра, а это означает неминуемую остановку всех контейнеров, запущенных в данной системе. В отдельно взятой виртуальной машине можно производить любые изменения ядра ОС без опасения навредить процессам, запущенным в соседних ВМ.

Безопасность

Каждый производитель или разработчик нахваливает своё решение. Зачастую они используют не слишком веские доводы или апеллируют к общим принципам, забывая о важных нюансах или умышленно умалчивая о них.

Принято считать, что полноценная виртуализация позволяет обеспечить существенно более надёжную изоляцию между гостевыми системами, нежели контейнеры. Основной довод, приводимый апологетами данной позиции: благодаря совместно используемому ядру ОС, процесс, «убежавший» из своего контейнера, может как минимум навредить всей системе в целом, помешав нормальной работе ядра ОС. Или даже, получив доступ к данным, описывающим другие процессы в системе, пробраться в соседний контейнер и выполнить в нём какие-то злонамеренные действия.

На первый взгляд, полноценные виртуальные машины лишены этого недостатка, так как используемое гостевыми системами ПО полностью изолировано от других гостей. Но если взглянуть на систему виртуализации более внимательно, то обнаружится гипервизор, управляющий всеми гостевыми системами. То есть можно представить себе ситуацию, когда злонамеренный процесс «совершит побег» из своей ВМ и проникнет в гипервизор благодаря уязвимости в коде гипервизора или даже некоторым особенностям аппаратуры процессора. newpage К сожалению, это вовсе не теоретические рассуждения. Достаточно взглянуть на неполный список известных побегов из ВМ, чтобы оценить масштабы бедствия. Разнообразные гипервизоры оказываются жертвами, и проблема со временем никуда не девается. Более того, о многих имеющихся до сих пор уязвимостях мы, простые пользователи и даже исследователи компьютерной безопасности, ещё не знаем.

Таким образом, вопрос о более безопасном решении до сих пор остаётся открытым. Думается, что более безопасной систему может сделать комплекс технологий и процессов, обеспечивающих снижение рисков по всем возможным направлениям, а не попытка использовать лишь одну «магическую» технологию.

Однако стоит отметить, что в реальной жизни контейнеры обычно запускаются внутри виртуальных машин, которые заменяют собой физические машины. Так осуществляется высокая изоляция контейнеров и сохраняется высокая эффективность использования ресурсов контейнерами. Характерный пример — арендованный виртуальный сервер, на котором запущены разнообразные сервисы, реализованные в контейнерах. В настоящее время обычно речь идёт о виртуальной машине KVM.

Примеры реализации

Virtuozzo и OpenVZ

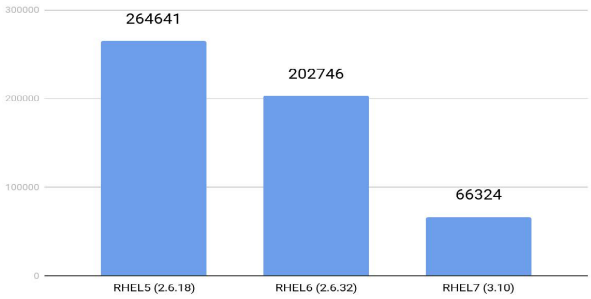
Virtuozzo и OpenVZ (сокращение от Open Virtuozzo) — это продукты компании Parallels, позволяющие на одной физической машине создавать множество виртуальных окружений (virtual environments). В современной терминологии эти продукты можно было бы назвать контейнерами операционных систем, так как упомянутые виртуальные окружения содержали внутри полноценные Linux-дистрибутивы. Похоже, что даже термин Virtual Private Server (VPS) появился в своё время благодаря Virtuozzo/OpenVZ.

Разработка Virtuozzo началась в прошлом тысячелетии, в 1999 году, если быть более точными. Его разработали инженеры компании SWsoft, в 2007 году переименованной в Parallels, на основе ядра ОС Linux версии 2.2. А в январе 2002 года был выпущен первый коммерческий продукт — Virtuozzo 2.0, но уже на основе ядра Linux версии 2.4.0 test1.

В 2005 году было принято решение открыть все наработки команды Parallels в ядре Linux, а также необходимый минимум утилит управления. Так появился проект OpenVZ. Поскольку основой Virtuozzo было ядро Linux, в тот момент не обладавшее средствами создания и управления виртуальными окружениями, то все изменения относительно ванильного ядра приходилось адаптировать к новым выпускам ядра Linux, которое весьма существенно изменяется от версии к версии. То есть только для обновления версии используемого ядра Linux приходилось затрачивать существенные инженерные ресурсы. Термин «контейнер» тогда ещё не был в ходу, он появился в названии продукта Solaris Containers компании Sun Microsystems в январе 2005 года.

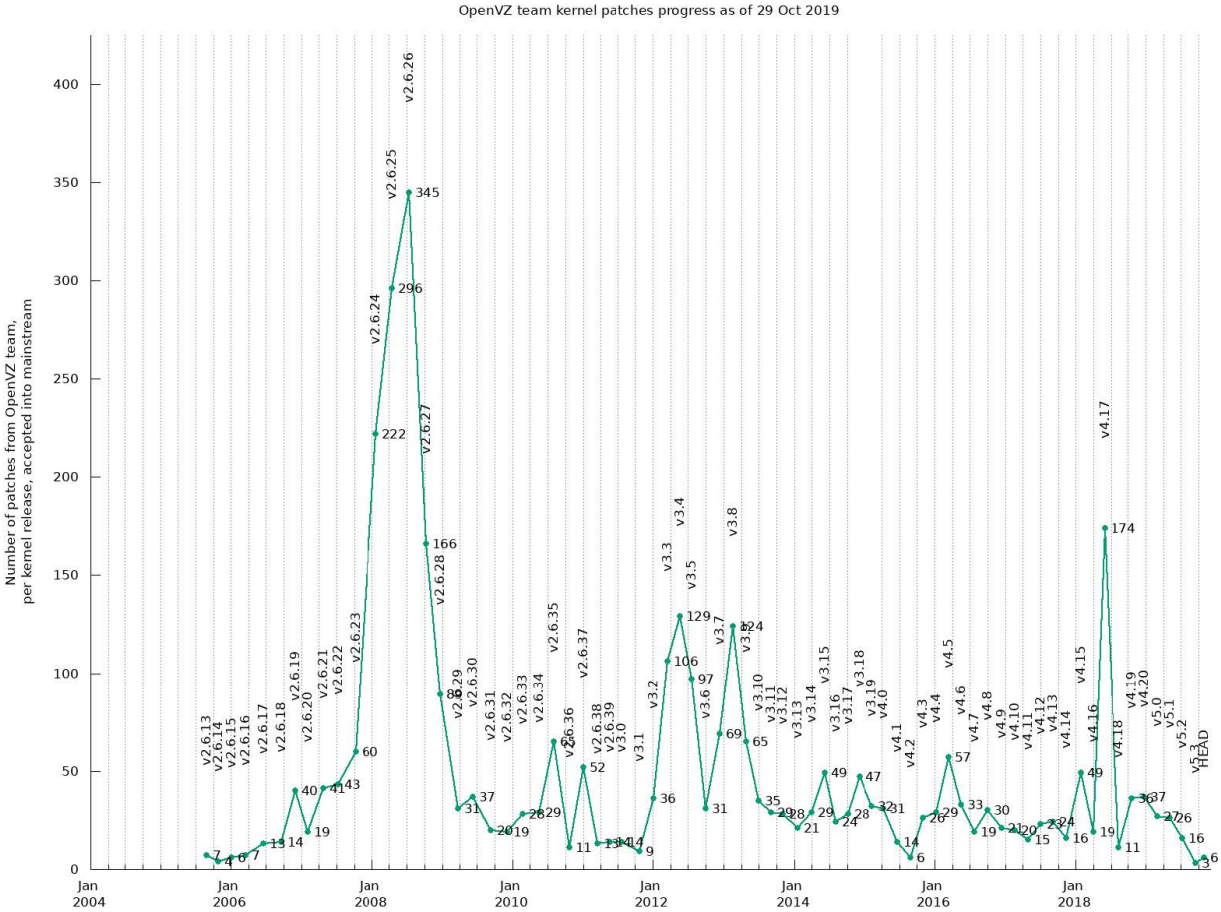
Ванильное ядро (vanilla kernel, mainline kernel) — термин, которым называют оригинальную версию ядра Linux, в которую не было внесено изменений по сравнению с kernel.org. Вероятно, такое значение («стандартный», «вариант по умолчанию») у слова «ванильный» появилось благодаря популярности ванильного пломбира, который подразумевается по умолчанию, если речь идёт о самом простом мороженом.

Решение этой проблемы состоит в интеграции своих изменений в основной код ядра. Благодаря этому в более новых ядрах необходимый код уже будет присутствовать. Так наработки инженеров Parallels оказали существенное влияние на развитие технологии Linux-контейнеров. В частности, такие компоненты, как PID namespaces, network namespaces, memory cgroups, Checkpoint/Restore In Userspace (CRIU) и многие другие были либо разработаны, либо усовершенствованы инженерами Parallels. Подробности можно узнать в разделе FAQ проекта OpenVZ на «Википедии».



Количество изменённых строк кода в ядре Virtuozzo относительнованильного ядра Linux

Ниже вы видите график, иллюстрирующий количество изменений, сделанных инженерами компании Parallels и принятых в основной код ядра Linux.



Позже продукты OpenVZ и Virtuozzo были объединены в продукт, взявший имя открытого проекта OpenVZ.

В настоящее время разработка OpenVZ, похоже, сводится к выпуску минорных версий ядра из состава Red Hat Enterprise Linux 6 для OpenVZ 6. Решения на базе OpenVZ/Virtuozzo можно найти лишь в давно исполь-

зуемых системах, которые ещё не были или уже никогда и не будут переведены на использование более современных технологий. Тем не менее, благодаря успешным продуктам для создания виртуальных окружений, появились и существенно развились удобные, безопасные и эффективные инструменты для создания контейнеров и управления ими.

Интересно отметить, что OpenVZ/Virtuozzo в основном нашли применение на рынке хостинга веб-сайтов, где долгое время невероятно популярный VPS-хостинг был синонимом OpenVZ/Vituozzo. Так произошло во многом благодаря низкой стоимости, которая стала возможной благодаря запуску множества виртуальных окружений на одном физическом сервере. Однако с появлением более мощных серверов и удобной и эффективной виртуализации на базе KVM всё больше хостинг-провайдеров стали переходить на предоставление клиентам экземпляров виртуальных машин вместо контейнеров. Видя спрос на решения на базе KVM, компания Parallels в последнем релизе OpenVZ добавила поддержку управления виртуальными машинами поверх гипервизора KVM.

Тем самым она сделала OpenVZ универсальным инструментом для управления виртуальными окружениями обоих типов: контейнерами и полноценными виртуальными машинами.

LXC и LXD

Именно благодаря наработкам в ядре Linux, сделанным для OpenVZ/Virtuozzo, стало возможным появление нового инструментария для создания контейнеров и управления ими. Он был назван незамысловато — Linux Containers, сокращённо LXC. Интересно, что в отличие от OpenVZ/Virtuozzo, LXC использует функциональность ванильного ядра Linux без каких-либо дополнительных изменений, а потому позволяет использовать практически любой современный Linux-дистрибутив для создания или запуска соответствующих контейнеров. Как и его предшественники (парочка OpenVZ/Virtuozzo), LXC создаёт виртуальное окружение, предназначенное для запуска полноценной ОС, — разумеется, лишь его часть в пространстве пользователя, без ядра ОС. По сути, LXC можно было бы назвать примерной реализацией системных контейнеров в Linux.

Пошаговую инструкцию для начинающих можно найти на сайте Linux Containers. Важно отметить, что, будучи достаточно низкоуровневым инструментарием, LXC требует выполнения пользователем относительно большого количества настроек хозяйской системы по сравнению с инструментами, которые будут рассмотрены позже. Но есть в этом и красота инструментария в духе UNIX: простые утилиты хорошо выполняют простой набор операций, а более сложные задачи решаются путём запуска

разных утилит пользователем.

Поскольку работа с низкоуровневыми утилитами не всегда удобна, особенно при использовании множества контейнеров и сложных сценариях управления, спустя некоторое время был разработан новый, более высокоуровневый, инструментарий — LXD. Читается «лекс-ди», это сокращение от Linux Container Daemon. Его разработчики, в основном инженеры компании Canonical, называют LXD гипервизором контейнеров, делая отсылку к диспетчеру виртуальных машин, также именуемому гипервизором. Хотя, разумеется, никакой это не гипервизор, а просто диспетчер контейнеров.

В случае диспетчера виртуальных машин название «гипервизор» вполне уместно, так как оно противопоставляется «супервизору», в качестве которого выступает ядро ОС. И гипервизор оказывается как бы супервизором супервизоров, отсюда и «гипер-». При использовании контейнеров диспетчер контейнеризованных окружений не управляет супервизорами этих окружений, потому как все эти окружения используют ядро хозяйской системы, поверх которого работает и сам диспетчер. Так что «гипервизор контейнеров» — это в определённом смысле преувеличение или метафора.

Ещё более ёмко и точно LXD описывается в GitHub-репозитории, где ведётся его разработка: «демон, использующий liblxc и предоставляющий REST API для управления контейнерами». Кроме демона, который управляет непосредственно контейнерами, разрабатывают и удобный клиент для простого взаимодействия с демоном. Доля иронии заключается в том, что клиент для lxd носит название lxc, хотя его последняя буква — сокращение от client, а не container.

Благодаря такой клиент-серверной архитектуре возможно удалённое управление демоном, то есть совсем не обязательно запускать клиент управления на той же машине, что и демон. Это сделало возможным создание клиентов в том числе для ОС Windows и даже macOS.

Хочется надеяться, что мы помогли вам разобраться, чем же различаются LXC и LXD. Тем не менее рекомендуем начать своё знакомство с Linux Containers именно с LXD. Это более удобный и простой в освоении инструментарий. Инструкции для начала работы с LXD, в том числе и на русском языке, доступны на официальном сайте проекта.

Более того, можно воспользоваться облачным сервисом проекта и получить опыт работы с LXD без подготовки своей собственной системы.

Имея хозяйскую систему с дистрибутивом Ubuntu 16.04, запустим контейнер с CentOS 7:

```
root@tryit-welcomed:~# lxc launch images:centos/7 centos-7
root@tryit-welcomed:~# lxc exec centos-7 -- /bin/bash
```

И можем наблюдать следующее внутри контейнера:

```
[root@centos-7 ~]# ps -aef --forest
UID        PID  PPID  C STIME TTY          TIME CMD
root       316     0  0 20:35 ?        00:00:00 /bin/bash
root       338   316  0 20:41 ?        00:00:00  \_ ps -aef --forest
root         1     0  0 20:35 ?        00:00:00 /sbin/init
root        35     1  0 20:35 ?        00:00:00 /usr/lib/systemd/systemd-
   journald
root        36     1  0 20:35 ?        00:00:00 /usr/lib/systemd/systemd-
   udevd
root        58     1  0 20:35 ?        00:00:00 /usr/lib/systemd/systemd-
   logind
dbus        59     1  0 20:35 ?        00:00:00 /usr/bin/dbus-daemon --
   system --address=systemd: --nofork --nopidfile --systemd-activation
root        71     1  0 20:35 console  00:00:00 /sbin/agetty --noclear --
   keep-baud console 115200,38400,9600 linux
root        72     1  0 20:35 ?        00:00:00 /usr/sbin/crond -n
root       250     1  0 20:35 ?        00:00:00 /sbin/dhclient -1 -q -lf /
   var/lib/dhclient/dhclient--eth0.lease -pf /var/run/dhclient-eth0.pid -
   H cento
root       311     1  0 20:35 ?        00:00:00 /usr/sbin/rsyslogd -n
```

Вывод утилиты ps — она отображает запущенные в системе процессы — стоит запомнить для сравнения с тем, что мы увидим при использовании контейнеров приложений:

```
[root@centos-7 ~]# cat /etc/centos-release
CentOS Linux release 7.7.1908 (Core)
[root@centos-7 ~]# uname -r -v
4.4.0-170-generic #199-Ubuntu SMP Thu Nov 14 01:45:04 UTC 2019
```

Также интересно обратить внимание на содержимое файла /etc/centos-release и даже сам факт его наличия. Сравните его с полным названием версии ядра ОС. Легко обнаружить, что запущен дистрибутив CentOS версии 7.7, однако ядро ОС вовсе не из этого дистрибутива, а из хозяйской системы с Ubuntu 16.04. Linux-дистрибутив CentOS 7, как и его двойник Red Hat Enterprise Linux 7, использует ядро Linux версии 3.10.

Интересные особенности LXC/LXD:

1. Наличие реестра заранее подготовленных образов почти всех популярных дистрибутивов. Запуск нового контейнера производится быстро и просто: как мы видели в примере выше, достаточно всего двух команд.

2. Если LXC как низкоуровневый инструментарий для создания контейнеров и управления ими доступен в любом современном Linux-дистрибутиве, то LXD — в каком-то смысле удобная надстройка над LXC — в основном доступен в Ubuntu. Это не удивительно, так как в основном именно инженеры Canonical занимаются разработкой LXD. Однако благодаря системе управления пакетами Snappy LXD возможно установить и в прочих дистрибутивах.

Snappy — система развёртки и управления пакетами, также разработанная Canonical для мобильной Ubuntu. Инструкция по установке snapd

(собственно, диспетчера пакетов Snappy) доступна на сайте snapcraft.io. Однако стоит понимать, что пакеты Snappy, будучи сторонними по отношению к своей системе управления пакетами дистрибутивов, отличных от Ubuntu, могут давать несколько неожиданный результат. Мы не говорим о производных от Ubuntu вроде Mint и прочих.

3. LXC — это набор утилит, а LXD — это в первую очередь демон, использующий liblxc, то есть, по сути, набор инструментария LXC. Не путайте с клиентом LXD под названием lxc.

4. Будучи полноценным Linux-дистрибутивом, LXC-контейнеры позволяют запускать контейнеры внутри себя, тем самым реализуя вложенную контейнеризацию. Не то чтобы это где-то широко использовалось, но сама по себе возможность есть. Для сравнения, контейнеры приложений, которые мы рассмотрим в дальнейшем, на такое принципиально не способны.

5. В LXD версии 3.18, выпущенном в октябре 2019 года, разработчики заявили о поддержке не только LXC-контейнеров, но также и полноценных виртуальных машин. Можно будет выбирать тип изоляции (контейнер или виртуальная машина) передачей одного параметра при использовании стандартного клиента lxc или при использовании непосредственно REST API, предоставляемого демоном LXD.

Забавно наблюдать повторение пути, пройденного инженерами компании Parallels. Мы помним, как их продукт OpenVZ 7.0, объявленный в далёком уже 2016 году, преподносился как универсальное решение для виртуализации. На выбор пользователя предлагалось использование виртуального окружения Virtuozzo или полноценной ВМ на основе KVM.

Docker

Мы поговорили о LXC и LXD как о правильно реализованной технологии Linux-контейнеров в их первоначальном смысле — как о виртуализованном дистрибутиве, изолированном от хозяйской системы и себе подобных. Имеется в виду использование только той функциональности, которую предоставляет ванильное ядро, без «секретного соуса» отдельных разработчиков или компаний. Теперь стоит познакомиться с Docker.

Первое упоминание Docker, продукта компании dotCloud, прозвучало в выступлении основателя Docker Inc. (в тот момент ещё dotCloud Inc.) Соломона Хайкса на конференции PyCon 2013.

На сегодняшний день Docker — практически синоним почти всего, что связано с контейнерами. По крайней мере, у обывателя может возникнуть такое ощущение. Хотя в конце 2019 года при более внимательном взгляде становится понятно, что помимо Docker существует ещё множество продуктов и решений, которые серьёзно угрожают компании Docker Inc. и их продуктам. Подробнее об этом вы можете прочитать в статьях Part II: Why Is There No Docker in OpenShift 4 and RHEL 8? и What We Announced Today and Why it Matters (Mirantis приобретает Docker Enterprise platform у Docker Inc.).

Так что же такое Docker? Первое, с чем стоит определиться: Docker Inc. — это компания, занимающаяся разработкой одноимённого продукта. Продукты компании Docker Inc. — это средства для создания и распространения контейнеров, а также управления ими. В частности, Docker Engine, Docker Compose, Docker Desktop for Mac/Windows, в конце концов, Docker Hub.

Отметим, что Docker Engine до версии 0.9 использовал LXC, инструменты которого в основном реализованы на языке Си, как средство создания и запуска контейнеров. Начиная с версии 0.9, Docker использует свой движок libcontainer, полностью написанный на языке программирования Go. Более того, Docker Engine первоначально представлял собой один исполняемый файл /usr/bin/docker, который содержал функциональность демона, клиента, средства построения контейнеров, а также клиента для работы с реестрами образов. Так было проще разрабатывать и распространять ещё не очень сложный программный продукт.

Существенные изменения произошли в версии 1.11. Во-первых, разработчики разделили клиента и демона, теперь это стал /usr/bin/dockerd. Более того, уже в версии 1.12 были вынесены в отдельные модули среды исполнения контейнеров runC и низкоуровневый инструмент для взаимодействия с абстрактной средой исполнения контейнеров containerd.

И чем дальше, тем больше происходит разделение ранее монолитного инструмента на множество модулей. Причём в соответствии со своим Infrastructure Plumbing Manifesto компания Docker будет стараться:

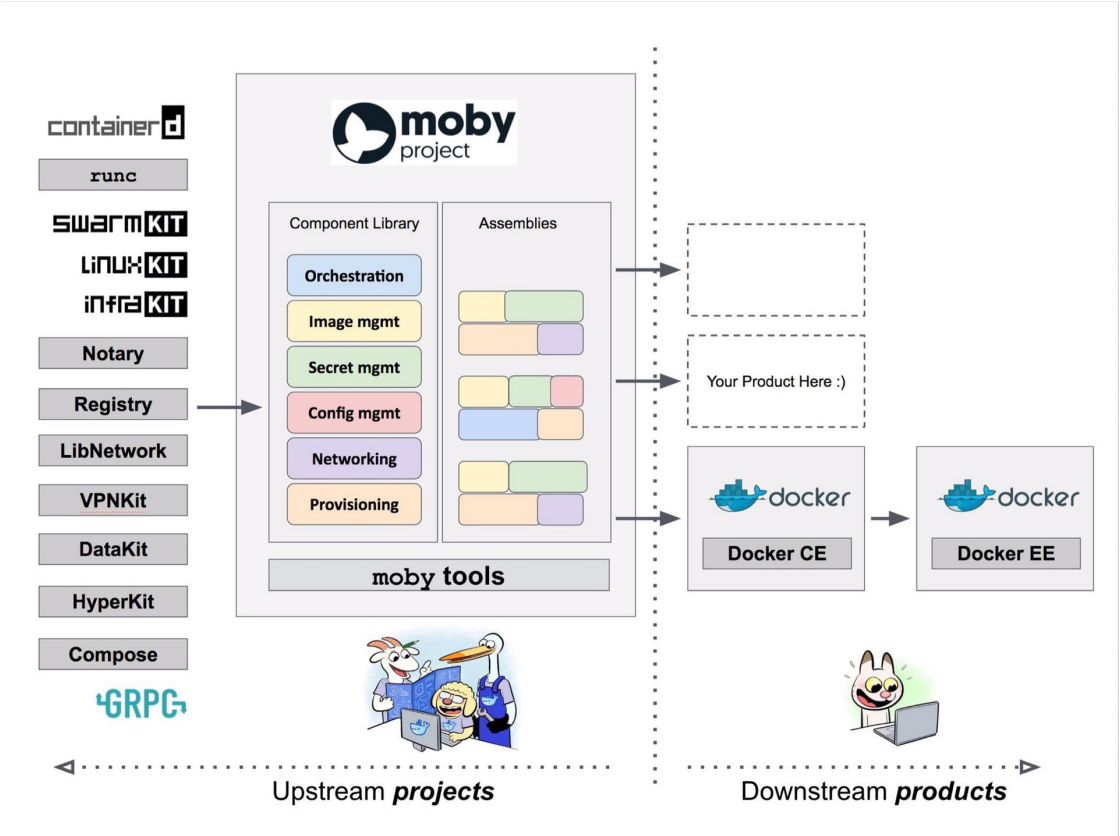
1. Всегда, если это возможно, повторно использовать уже имеющиеся компоненты и отдавать назад свои улучшения. Если же действительно необходимо создать новый компонент, то делать его лёгким в повторном использовании и предусмотреть возможность принятия улучшений от сторонних разработчиков.

2. Следовать принципам UNIX: несколько простых компонентов лучше, чем один, но сложный.

3. Определять стандартные интерфейсы, которые позволят с лёгкостью

связать несколько простых компонентов в одну сложную систему.

Сегодня мы видим результаты следования этим принципам: инструментарий Docker стал глубоко модульным, что даёт возможность использовать только те его части и в том виде, который наиболее удовлетворяет требованиям конкретного случая. Апофеоз этой модульной структуры — проект Moby. При помощи Moby, как из кирпичиков Lego, можно построить как Docker, так и свою собственную систему управления контейнерами.



Также стоит отметить, что модульность стала возможна благодаря стандартизации некоторых компонентов в рамках проекта Linux Foundation под названием Open Container Initiative (OCI). В частности, были разработаны стандарты для среды исполнения и образов контейнеров. Причём в обоих случаях наработки Docker Inc. послужили основой для стандартов. Стоит упомянуть лишь runC, который, будучи изначально частью Docker engine, претерпел небольшие изменения и стал базовой реализацией среды выполнения контейнеров.

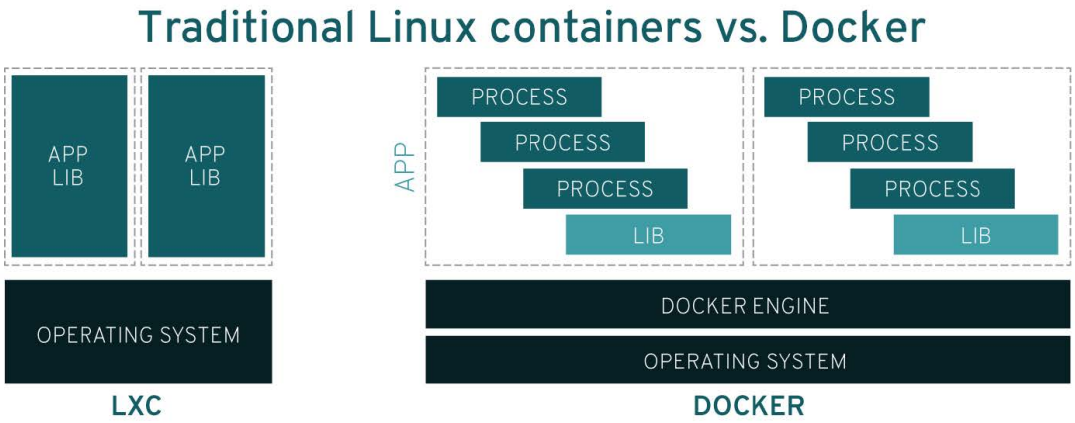
Фактически стандартизация формата образов и среды выполнения привела к тому, что появилась возможность использовать одни и те же образы с самыми разнообразными средами запуска. Более того, это дало толчок бурному развитию систем запуска контейнеров: CRI-O, rkt, runhcs, kata-runtime. Причём некоторые среды запуска вышли за пределы исходного определения контейнеров. Они используют аппаратные средства уско-

рения виртуализации для создания более безопасных окружений — легковесных виртуальных машин. Примеры таких окружений: Kata containers и контейнеры Windows при использовании Hyper-V изоляции. Не путайте с Docker-desktop, но об этом позднее.

Возвращаясь к Docker, поговорим о его особенностях по сравнению с ранее рассмотренными Virtuozzo/OpenVZ и LXC.

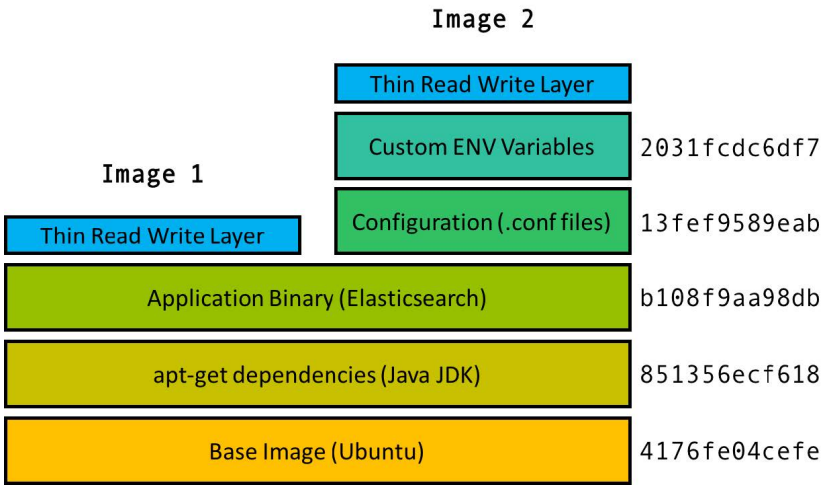
Контейнер приложения

Важное отличие от ранее существовавших систем контейнеризации — смещение фокуса с эмуляции полноценной системы к легковесному контейнеру, содержащему только одно приложение и все требующиеся ему зависимости. Если использовать контейнер не как альтернативу виртуальной машине у хостинг-провайдера, а как способ упаковки одного сервиса или приложения с необходимым окружением, то можно выкинуть из такого контейнера всё лишнее. Можно сократить объём памяти, занимаемый контейнером и его образом, и использование ресурсов процессора за счёт исключения запуска служебных процессов: разнообразных демонов и даже системы инициализации.



Read Hat Containers: What is DOCKER?

Многослойные образы контейнеров



Слои Docker

Второе важное отличие — это многослойность. Преимущество перед монолитными образами состоит в том, что теперь каждый из слоёв можно многократно использовать в других, более сложных, образах. Безусловно, базовый образ — это, как правило, минималистичная файловая система какого-то из популярных Linux-дистрибутивов, и она весьма монолитна. Но и то не совсем:

```
$ docker history ubuntu:bionic
IMAGE           CREATED       CREATED BY SIZE
549b9b86cb8d    10 days ago   /bin/sh -c #(nop)  CMD ["/bin/bash"] 0B
<missing>       10 days ago   /bin/sh -c mkdir -p /run/systemd && echo 'do
   ... 7B
<missing>       10 days ago   /bin/sh -c set -xe   && echo '#!/bin/sh' >
   /... 745B
<missing>       10 days ago   /bin/sh -c [ -z "$(apt-get indextargets)" ]
   987kB
<missing>       10 days ago   /bin/sh -c #(nop) ADD file:53f100793e6c0adfc
   ... 63.2MB
```

Для сравнения, вот так выглядит более сложный образ Baseimage-docker:

```
$ docker history 166cfc3f6974
IMAGE           CREATED          CREATED BY SIZE
166cfc3f6974    23 months ago    /bin/sh -c #(nop) CMD ["/sbin/my_init"] 0B
<missing>       23 months ago    /bin/sh -c #(nop) ENV DEBIAN_FRONTEND=
   telety... 0B
<missing>       23 months ago    /bin/sh -c /bd_build/prepare.sh &&  /
   bd_buil... 97.3MB
<missing>       23 months ago    /bin/sh -c #(nop) COPY dir:
   e7a5eda59d878c69c... 39.4kB
<missing>       23 months ago    /bin/sh -c #(nop) MAINTAINER Phusion <
   info@p... 0B
<missing>       23 months ago    /bin/sh -c #(nop)  CMD ["/bin/bash"] 0B
```

```
<missing>       23 months ago   /bin/sh -c mkdir -p /run/systemd && echo '
    do... 7B
<missing>       23 months ago   /bin/sh -c sed -i 's/^#\s*\(deb.*universe
    \)$... 2.76kB
<missing>       23 months ago   /bin/sh -c rm -rf /var/lib/apt/lists/* 0B
<missing>       23 months ago   /bin/sh -c set -xe   && echo '#!/bin/sh' >
     /... 745B
<missing>       23 months ago   /bin/sh -c #(nop) ADD file:
    a3344b835ea6fdc56... 112MB
```

Легко заметить большое количество изменений по сравнению с базовым образом Ubuntu 18.04.

Новый слой автоматически создаётся при выполнении каждой команды Dockerfile при создании образа. В частности, вот так создавался образ, рассмотренный выше:

```
ARG BASE_IMAGE=ubuntu:18.04
FROM $BASE_IMAGE
MAINTAINER Phusion <info@phusion.nl>

ARG QEMU_ARCH
ADD x86_64_qemu-${QEMU_ARCH}-static.tar.gz /usr/bin

COPY . /bd_build

RUN /bd_build/prepare.sh && \
    /bd_build/system_services.sh && \
    /bd_build/utilities.sh && \
    /bd_build/cleanup.sh

ENV DEBIAN_FRONTEND="teletype" \
    LANG="en_US.UTF-8" \
    LANGUAGE="en_US:en" \
    LC_ALL="en_US.UTF-8"

CMD ["/sbin/my_init"]
```

Другие образы могут использовать его как свой базовый образ и так далее. Причём данные каждого образа хранятся на локальной файловой системе в единственном экземпляре. То есть при запуске десяти контейнеров, использующих один и тот же образ, на хозяйской системе будет храниться и использоваться только один экземпляр образа. Более того, если будет создан новый образ на основе имеющегося базового, то на диске хозяйской системы будет дополнительно сохранена только разница между базовым и новым образами. Полезно вспомнить, что сокращение объёма используемой памяти конвертируется в существенное ускорение работы системы. Это происходит благодаря тому, что большая часть данных может быть доступна в быстрой памяти, близкой к процессору: кешах процессора или, на худой конец, в ОЗУ, а не только на диске или в сети.

Поддержка различных хозяйских ОС

Как мы помним, Virtuozzo/OpenVZ и LXC (Linux Containers даже по определению) существовали только на хозяйских системах с ядром Linux, то есть фактически только в Linux-дистрибутивах. Однако компания Docker Inc. сумела преодолеть это ограничение.

Первая возможность использовать Docker поверх хозяйской ОС, отличной от Linux-дистрибутива, появилась благодаря использованию полноценной виртуальной машины, внутри которой был запущен, собственно, Linux-дистрибутив. Справедливости ради отметим, что такой подход был применим и к другим системам контейнеризации: любой желающий мог установить в виртуальной машине Linux-дистрибутив и в нём Docker.

Такой подход не слишком удобен, так как не позволяет управлять системой контейнеризации непосредственно из хозяйской системы. Все действия нужно производить, находясь внутри виртуальной машины. То же касается сложностей со взаимодействием с сервисами или приложениями внутри контейнеров: пользователю самому пришлось бы заниматься настройкой проброса портов из хозяйской системы в гостевую и т. д.

Компания Docker в своём решении Docker Toolbox on Windows (равно как и on Mac) сделала работу с контейнерами в виртуальной машине гораздо более удобной. Она предложила настроить всё за пользователя и предоставила удобный инструментарий специально для такого сценария:

1. Kinematic — графическая утилита для управления контейнерами.
2. Docker Machine — утилита командной строки для управления виртуальными окружениями и контейнерами в них.
3. Docker Quickstart — терминал, настроенный для работы с Docker Engine внутри виртуальной машины.
4. Виртуальная машина для Oracle VirtualBox с установленным легковесным Linux-дистрибутивом Boot2Docker.

Boot2Docker — легковесный Linux-дистрибутив, созданный на базе Tiny Core Linux. Работает будучи полностью загруженным в ОЗУ, имеет загружаемый образ размером 45 Мбайт. В настоящее время находится в режиме минимальной поддержки как запасной вариант для пользователей, которые по разным причинам не могут использовать более современное решение Docker for Windows или Docker for Mac. Это пользователи Windows 7, для которой Docker for Windows просто недоступен, и пользователи Windows 10 Home, в которой невозможно включить Microsoft Hyper-V.

Благодаря этим нововведениям значительно улучшился пользовательский опыт при работе с контейнерами из Windows или macOS. На это решение был высокий спрос, но его эффективность была низкой из-за необходимости запуска полноценной виртуальной машины, причём «чужой» для хозяйской ОС. В качестве виртуальной машины использовался Oracle VirtualBox — решение хоть и доступное на Windows и macOS, но тем не менее не очень интегрированное в хозяйскую ОС. Другого и нельзя было ожидать от продукта стороннего относительно хозяйской ОС производителя.

В результате миру были представлены более интегрированные с хозяйской ОС решения: Docker Desktop for Windows и Docker Desktop for Mac.

Docker for Windows

В Docker for Windows для запуска Linux-контейнеров используется встроенный гипервизор Microsoft Hyper-V. Под его управлением запускается минималистичная, хотя всё ещё полноценная, виртуальная машина со специально подготовленным Linux-дистрибутивом. Это собранный при помощи LinuxKit дистрибутив, что можно засвидетельствовать, обнаружив работающей виртуальную машину DockerDesktopVM.

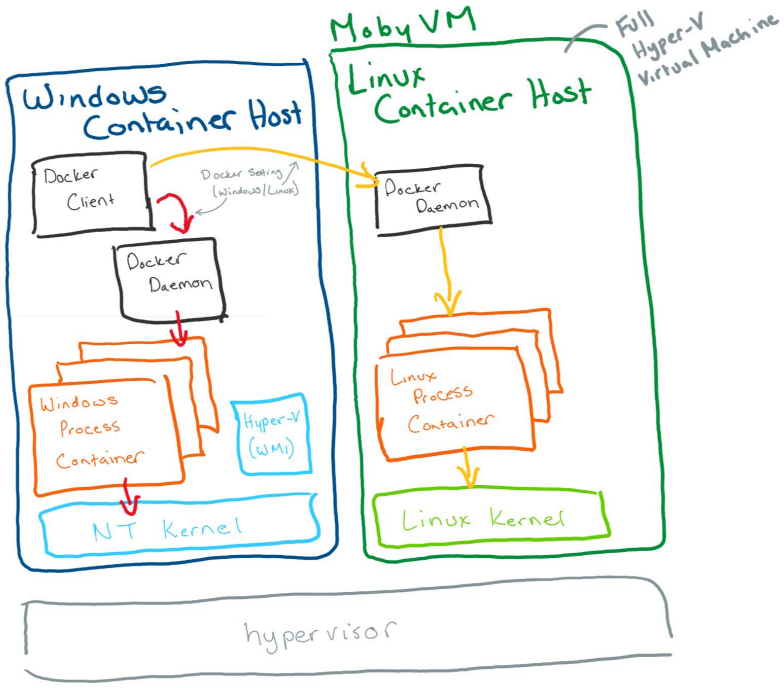
```
PS C:\> Get-VM

Name             State   CPUUsage(%)  MemoryAssigned(M)  Uptime     Status            Version
----             -----   -----------  -----------------  ------     ------            -------
DockerDesktopVM Running 0            2048               00:01:14   Работает нормально 9.0
```

LinuxKit — инструмент построения Linux-дистрибутивов, специально предназначенных для запуска систем контейнеризации.

Похоже, что до настоящего времени инженеры Docker Inc. не могут решить, как называть виртуальную машину. До сих пор мы видели Moby VM, MobyLinuxVM и DockerDesktopVM.

В отличие от более ранней реализации Docker Toolbox for Windows, в Docker Desktop for Windows часть инструментария запускается непосредственно в хозяйской ОС Windows. В случае использования полноценной виртуальной машины это клиент Docker (собственно, утилита docker). Тем не менее одна и та же виртуальная машина используется для запуска всех контейнеров, то есть все они используют одно и то же ядро Linux. Более того, поскольку Docker Engine находится внутри той же виртуальной машины, её приходится всё время держать включённой, иначе не удастся запустить контейнер в случае необходимости.



Устройство Docker for Windows с полноценной виртуальной машиной

Сейчас ведётся активная работа над новой реализацией Linux-контейнеров в Windows с названием LCOW (Linux Containers on Windows). Её существенные отличия: во-первых, каждый контейнер запускается в собственном виртуальном окружении и использует свою собственную копию ядра Linux, а во-вторых, всё управление контейнерами происходит на стороне хозяйской ОС. В том числе и Docker Engine остаётся в хозяйской системе, а виртуальные окружения создаются по запросу при необходимости. Интересно, что такие виртуальные окружения не видны в диспетчере Hyper-V, даже когда Linux-контейнер запущен на исполнение.

Для примера можно запустить новый контейнер с Ubuntu 18.04:

```
PS C:\> docker run --rm -i -t ubuntu bash
root@bf2f16d5b75b:/# uname -a
Linux bf2f16d5b75b 4.19.27-linuxkit #1 SMP Sun Mar 10 18:51:44 UTC 2019
    x86_64 x86_64 x86_64
GNU/Linux
root@bf2f16d5b75b:/# cat /etc/issue
Ubuntu 18.04.3 LTS \n \l
```

И попытаться обнаружить запущенную виртуальную машину:

```
PS C:\> Get-VM
Name              State   CPUUsage(%) MemoryAssigned(M) Uptime   Status             Version
----              -----   ----------- ----------------- ------   ------             -------
DockerDesktopVM   Off     0           0                 00:00:00 Работает нормально 9.0
```

Утилита Get-VM ожидаемо рапортует о наличии лишь остановленной виртуальной машины, которая использовалась ранее при запуске Linux-контейнера без LCOW. Зато при помощи утилит Get-ComputeProcess и

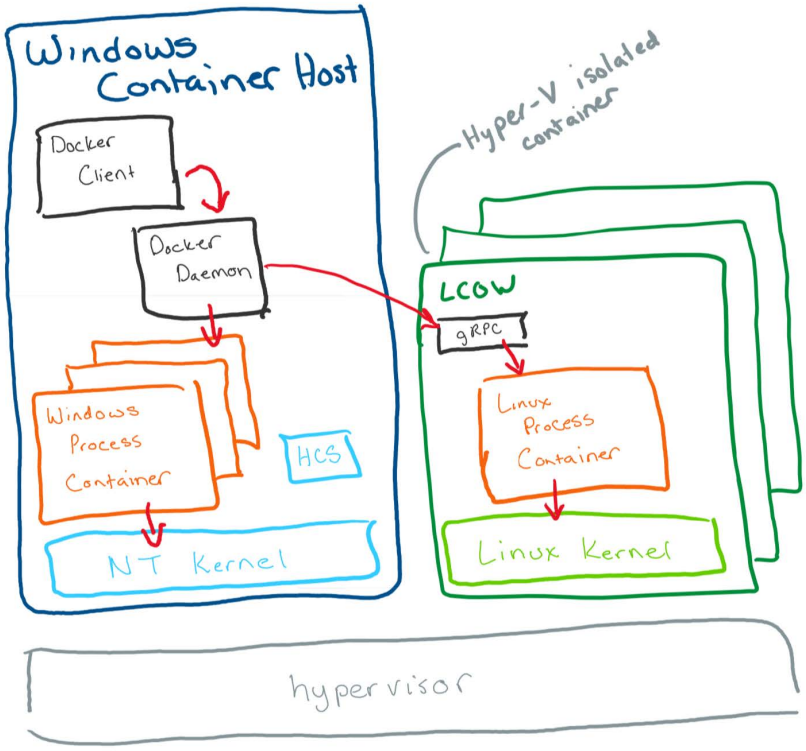
hcsdiag можно обнаружить наш контейнер. Смотрите объект с Id, совпадающим с Id Docker-контейнера:

```
PS C:\> Get-ComputeProcess
RunspaceId         : f43e60a7-8e89-4a49-8c72-20295e060568
Id                 : 9BE2B7E6-8724-4A60-A55B-DFF189F1919A
Type               : VirtualMachine
Isolation          : VirtualMachine
IsTemplate         : False
RuntimeId          : 9be2b7e6-8724-4a60-a55b-dff189f1919a
RuntimeTemplateId  :
RuntimeImagePath   :
Owner              : VMMS

RunspaceId         : f43e60a7-8e89-4a49-8c72-20295e060568
Id                 :
   bf2f16d5b75b351b7dfeb42c16f56741c35e2f442ccf287f2bbece4f587cbc93
Type               : Container
Isolation          : HyperV
IsTemplate         : False
RuntimeId          : cf4d055f-e68b-4f62-8f44-90ca03e9d592
RuntimeTemplateId  :
RuntimeImagePath   : C:\Program Files\Linux Containers
Owner              : docker

PS C:\> hcsdiag list
9BE2B7E6-8724-4A60-A55B-DFF189F1919A
VM, Created, 9BE2B7E6-8724-4A60-A55B-DFF189F1919A,
VMMS

bf2f16d5b75b351b7dfeb42c16f56741c35e2f442ccf287f2bbece4f587cbc93
Hyper-V Linux Container, Running, CF4D055F-E68B-4F62-8F44-90CA03E9D592,
docker
```



Устройство Docker for Windows с LCOW

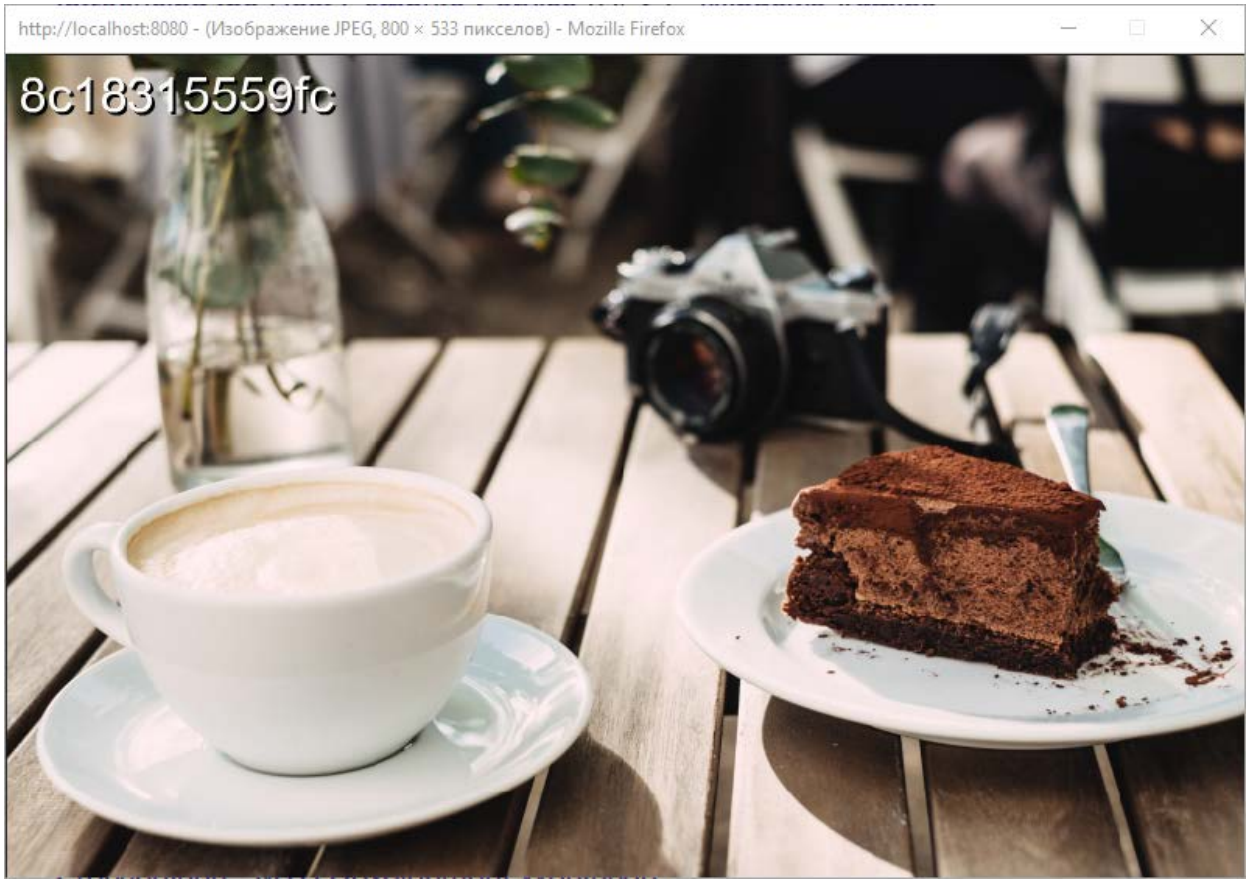
Интересно заметить, что реализация LCOW стала возможной благодаря ранее проделанной над Windows-контейнерами работе. Разработка Windows-контейнеров велась совместными усилиями инженеров Microsoft и Docker Inc. Одним из результатов работы над Windows Server Containers стал метод изоляции при помощи гипервизора Microsoft Hyper-V, который затем был использован для запуска Linux-контейнеров в легковесных виртуальных окружениях Host Compute Service (HCS).

Ещё одно преимущество использования LCOW — возможность одновременного запуска Linux- и Windows-контейнеров. Это наглядно представлено примером ниже:

```
PS C:\> docker run -d -p 8080:8080 --isolation=process chocolateyfest/
    appetizer:1.0.0
8c18315559fc8aab0c67723c12fa6666ba224f738c8f65ab2028f64f6caebaf6

PS C:\> start http://localhost:8080

PS C:\> docker container ls
CONTAINER ID IMAGE                              COMMAND       CREATED
    STATUS PORTS             NAMES
8c18315559fc chocolateyfest/appetizer:1.0.0 "node app.js" 37 seconds ago
    Up 37 seconds 0.0.0.0:8080->8080/tcp   eager_faraday bf2f16d5b75b
    ubuntu "bash" 32 minutes ago Up 32 minutes  clever_brown
```



В заключение разговора о Docker на Windows хочется заметить, что после удачного старта Windows-контейнеров на базе Docker инженеры Microsoft начали работы по выделению среды запуска контейнеров в независимый внешний компонент. Так появился runhcs. По сути, runhcs — это видоизмененный runC для работы в Windows и запуска контейнеров при помощи HCS. Это открывает дорогу на платформу Windows для других высокоуровневых систем управления контейнерами.

Docker for Mac

Docker for Mac использует подход с запуском Linux-дистрибутива, построенного при помощи LinuxKit под управлением гипервизора HyperKit, который разработали инженеры Docker Inc специально для работы с Docker. По сути, это мало чем отличается от Docker for Windows с MobyVM, то есть без использования LCOW. Весь Docker Engine и все контейнеры запускаются в одной перманентно запущенной виртуальной машине, что определённо не слишком удобно.

Существенное отличие Docker for Mac от Docker for Windows состоит в том, что на сегодняшний день не существует Mac-контейнеров. То есть на Mac возможно использование только Linux-контейнеров.

Однако стоит иметь в виду, что использование контейнеров в неродном окружении, то есть не в Linux-дистрибутиве, являющемся хозяйской ОС, чревато шероховатостями: более низкой эффективностью использования вычислительных ресурсов системы, существенным снижением скорости работы контейнеризованных приложений и т. д. То есть использование систем с Windows или macOS для «боевого» запуска приложения в контейнерах не имеет смысла. Тем не менее Docker for Windows/Mac даёт возможность заниматься разработкой, связанной с использованием контейнеров, в привычном для инженера окружении.

Docker: заключение

Мы поговорили об инструментах, разработанных компанией Docker Inc. Всё началось с использования доступного к тому моменту инструментария Linux Containers (LXC). Затем случилась первая существенная переработка Docker Engine с целью более эффективного решения новой задачи — контейнеризации одного приложения или службы в противовес полной ОС. Следующая существенная переработка Docker Engine привела к разделению монолитного исполняемого файла на несколько самодостаточных сущностей, каждая из которых решала свою конкретную задачу. Более того, некоторые из них легли в основу стандартов, разработанных Open Container Initiative (OCI). А далее в игру вступили другие разработчики и даже компании, такие как IBM, Red Hat (ныне поглощенная IBM), Google и т. д. Используя стандартные компоненты, они стали строить всё более совершенные и функциональные системы, использующие контейнеры.

После Docker

Можно смело сказать, что Docker Inc. создала современный рынок решений, связанных с использованием контейнеризованных окружений. Однако, как это иногда случается, звёздный час Docker Inc., похоже, остался в прошлом. Инициативу перехватили другие игроки, и будущее компании Docker Inc вызывает всё больше вопросов. Такие выводы можно сделать, опираясь на следующие факты:

1. Компания Docker Inc. всё ещё не приносит дохода и всецело зависит от внешних инвестиций. Исполнительный директор Docker Inc. в сентябре 2019 года разослал письмо сотрудникам о работе с двумя потенциальным инвесторами, которые могут обеспечить достаточное финансирование в будущем.

2. Mirantis приобретает Docker Enterprise у Docker Inc.

3. Red Hat отказалась от использования Docker в OpenShift 4 и Red Hat Enterprise Linux 8 в пользу своей собственной разработки Podman/Buildah/Skope

Так что будем с интересом смотреть в будущее и наблюдать за рождением новых продуктов и технологий если не от Docker Inc., то от кого-то другого.

Тем не менее именно благодаря Docker появились стандарты на образы контейнеров и сред их выполнения. Это открыло дорогу к созданию инструментов для управления контейнерами на любой вкус. Так появились:

- CRI-O и Podman/Buildah/Skopeo от Red Hat как альтернатива управления контейнерами без демона dockerd;

- масса окружений для запуска контейнеров: crun, runv, runhcs.

crun — это runC, переписанный на языке Си.

runV — среда запуска контейнеров, использующая средства гипервизоров.

runhcs — порт runC для Microsoft Host Compute Services (HCS).

Так что стоит ожидать развития имеющихся и появления новых инструментов, каждый из которых либо лучше решает уже знакомые задачи, либо нацелен на решение доселе неизвестных задач. Кто бы мог подумать, что технологии, использующие базовые принципы и инструменты ОС с ядром Linux, будут реализованы как родные на совершенно чужеродной платформе Windows!

На этой радостной ноте можно было бы закончить разговор о контейнерах, но хочется упомянуть ещё один интересный феномен — использование средств аппаратной виртуализации для запуска контейнеров.

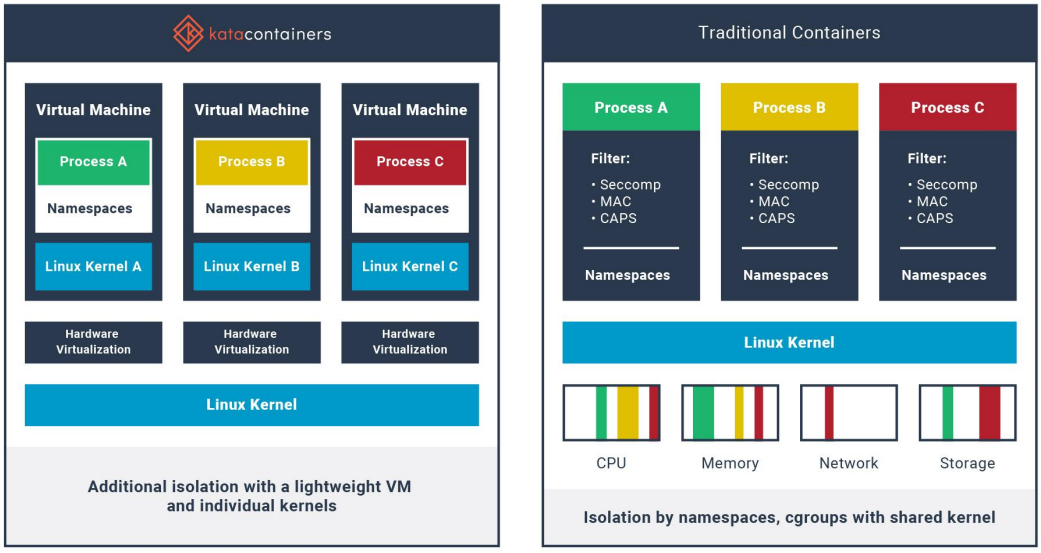
Контейнеры в легковесных виртуальных машинах

Ещё говоря о Docker for Windows, мы узнали, что запуск Windows-контейнеров может осуществляться в одном из двух окружений:

- с изоляцией процессов друг от друга средствами ОС (аналог идеи контейнеров в Linux);

- с изоляцией при помощи гипервизора Hyper-V (этот же режим используется LCOW).

Использование аппаратных средств поддержки виртуализации при изоляции разных окружений сулит большую безопасность, если говорить о возможности процессов из одного окружения проникнуть в другое. Хотя мы помним, что случается всякое. А спрос на такую более надёжную изоляцию определённо существует: например, при использовании одного и того же физического сервера для запуска контейнеров разных хозяев. А если есть спрос, то появится и предложение. О решении на платформе Windows мы уже поговорили, но что-то очень похожее есть и для Linux-систем. И имя этому решению — Kata containers.



Сравнение Kata Containers с обыкновенными контейнерами в Linux

Kata containers используют сильно оптимизированный Linux-дистрибутив, запущенный внутри виртуальной машины. Поддержаны следующие гипервизоры: ACRN, QEMU/KVM, Firecracker/KVM и Cloud Hypervisor/KVM.

Кроме минимизации самого образа гостевой ОС, используются некоторые интересные оптимизации как в ядре гостевой системы, так и на стороне гипервизора. Например, DAX — механизм доступа к накопителю данных в обход страничного кеша. Также KSM (англ. kernel same-page merging, иногда трактуется как kernel shared memory) — технология ядра Linux, которая позволяет ядру объединять одинаковые страницы памяти между различными процессами или виртуальными гостевыми системами в одну для совместного использования; реализация дедупликации данных для оперативной памяти. Также для ускорения запуска контейнеров используются шаблоны виртуальных машин.

Таким образом действительно достигается скорость запуска и работы контейнеризованных задач намного выше, чем при ручном запуске контейнеров в обыкновенных виртуальных машинах. Более того, плотность

размещения Kata-контейнеров всё ещё гораздо ближе к таковой для обычных контейнеров — много выше обыкновенных виртуальных машин.

А если прибавить соответствие стандарту OCI runtime и поддержку стандартных OCI-образов, то получается действительно привлекательное решение. Его можно легко встроить в имеющуюся инфраструктуру управления контейнерами, обеспечив при этом повышенную безопасность.

Заключение

Хотя контейнерная революция ещё даже не думает заканчиваться, мы можем наблюдать длинный и интересный путь, проделанный системами управления контейнерами, начиная с Virtuozzo/OpenVZ и заканчивая Cata containers. В недалёком будущем мы увидим новые интересные продукты, технологии и даже области применения контейнеров вдобавок к тому, что мы имеем сегодня.

Можно смело констатировать, что контейнеры уже прочно вошли в жизнь многих инженеров, компаний и даже индустрий. А потому стоит как минимум быть с ними знакомыми и готовыми к их использованию.

Используемые источники

1. Containers and lightweight virtualization.
2. Механизмы контейнеризации: namespaces.
3. Механизмы контейнеризации: cgroups.
4. Visualizing Docker Containers and Images. Перевод: Образы и контейнеры Docker в картинках.
5. Bringing Docker To Windows Developers with Windows Server Containers.
6. A comprehensive introduction to Docker, Virtual Machines, and Containers.
7. Diving Through The Layers: Investigating runc, containerd, and the Docker engine architecture.
8. Container Runtimes.
9. Образы и контейнеры Docker в картинках.
10. Demystifying Docker Containers Support on Windows 10 and Windows Server 1709.
11. Under the Hood: Demystifying Docker For Mac CE Edition.
12. В AWS представили Firecracker — «микровиртуализацию» для Linux.
13. Обновление гипервизоров Intel Cloud Hypervisor 0.3 и Amazon Firecracker 0.19, написанных на Rust.
14. Выпуск гипервизора для встраиваемых устройств ACRN 1.2, развиваемого в Linux Foundation.
15. Kata Containers. The way to run virtualized containers.
16. Контейнеризация при помощи LXC.
17. Windows Server and Docker, The Internals Behind Bringing Docker and Containers to Windows.
18. Learning Windows Server Containers.

Практическое задание

1. Назовите преимущества и недостатки контейнеров по сравнению с виртуальными машинами.

2. Запустите контейнер с базовым образом Ubuntu 18.04 в системе управления контейнерами на ваш выбор и выполните в запущенном контейнере несколько простых команд: uname, ls, cat.

3*. Запустите систему управления контейнерами на ваш выбор внутри LXC-контейнера.