АРЭПО. Лекция №1.

Введение в DevOps. Системы виртуализации и облачные решения.

1. Вступление.

Рад видеть всех вас на первой лекции нашего нового экспериментального курса "Автоматизация разработки и эксплуатации ПО".

Кто-то меня уже должен знать, но на всякий случай представлюсь: Михаил Кучеренко, аспирант кафедры ИУ5, по совместительству руководитель команды инженеров обеспечения доступности корпоративной виртуальной инфраструктуры OpenStack в VK.

2. О чем курс?

Автоматизация разработки и эксплуатации программного обеспечения.
Надеюсь, тут прослеживается обобщение и переложение на русский популярного ныне термина DevOps.

Про методологию DevOps мы сейчас слышим буквально из каждого утюга. О том что это, зачем нужно, какие технологии и инструменты используются - мы и поговорим в рамках этого курса. Причем, больше с точки зрения инженера, нежели управленца. Управленческие аспекты создания и жизненного цикла ПО подробно рассматриваются в магистратуре, особенно в курсах Марии Валерьевны Виноградовой и Юрия Григорьевича Нестерова. DevOps это закономерное развитие от спиральной и итерационной модели разработки ПО.

Лекции будут максимально ориентированы на теорию, необходимую для выполнения практических лабораторных работ.

3. Зачем курс?

Соответственно для этого и вводится данный кус - для получения практических навыков работы с технологиями, которые де-факто становятся корпоративными стандартами при разработке и эксплуатации ПО. В курсе мы столкнемся с GNU/Linux, Docker, Docker-compose, Kubernetes, OpenStack, Git, Prometheus, Ansible, Terraform, PostgreSQL, MySQL и многим другим. Все это встретится вам во время работы в самых современных компаниях - подтверждаю это как DevOps и SR-инженер с 4-летним стажем работы в VK.

Предполагается, что вы уже знакомы с основами работы операционных систем на базе GNU/Linux из курсов Петра Степановича Семкина и у вас сложилось базовое понимание компьютерных сетей из курса Артема Ильича Антонова. После прохождения текущего курса вы станете счастливыми обладателями основных знаний, необходимых для начинающего DevOps/SR-инженера. Разумеется, подробно охватить все новейшие технологии будет просто невозможно, данная область не стоит на месте. Но дальше вы

сможете сами совершенствоваться в выбранных направлениях - на DevOps и SR инженеров есть большой спрос на рынке труда, сам ищу себе в команду несколько человек.

4. Структура курса.

Примерный состав лекций и лабораторных.

Лекция 1. Введение в DevOps. Системы виртуализации и облачные решения. (Михаил Кучеренко)
Лабораторная 1. Настройка виртуальной машины GNU/Linux. Bash.

Лекция 2. Контейнеризация, Docker, СУБД. (Михаил Кучеренко + Алексей Якубов)
Лабораторная 2. Практические навыки работы с Docker.

Лекция 3. Масштабируемость и отказоустойчивость docker-кластеров. Docker-compose. (Антон Балашов)
Лабораторная 3. Балансировка и docker-compose.

Лекция 4. Контроль версий, Git, CI/CD. (Дмитрий Аладин)
Лабораторная 4. Настройка CI/CD в GitLab.

Лекция 5. Мониторинг, журналирование и оповещение событий. (Михаил Кучеренко)
Лабораторная 5. Мониторинг: Prometehus + Grafana + Alertmanager.

Лекция 6. Конфигурационное управление. Ansible, Terraform. (Антон Балашов)
Лабораторная 6. Ansible: inventory, playbook, role.

Лекция 7. Kubernetes. (Алексей Якубов)
Лабораторная 7. Развертывание кластера Kubernetes.

Лекция 8. Облачная инфраструктура. BASIS. (Алексей Якубов)
Лабораторная 8. Практикум в BASIS.
...

5. Что такое DevOps?

А теперь наконец содержательная часть лекции.

> Интерактив: Поднимите руки кто знает, что такое DevOps.
> Я вот не знаю. И сомневаюсь, что знает кто-то, кроме ребят из Google, которые этот термин в оборот ввели.

Часто когда говорят про DevOps или SRE рисуют какие-то круги Эйлера, вписываю туда какие-то слова "разработка", "эксплуатация", "тестирование", а на пересечении ставят предмет нашего разговора. На мой взгляд очень плохая аналогия.

С этим термином есть большая проблема. После публикаций Google многие взялись внедрять методологию у себя. И выяснилось, что все понимают это по-разному. Кажется даже в стенах Google какое-то время не было окончательного понимания о том что же такое DevOps и SRE.

Термин DevOps состоит из двух английских слов:
• development – разработка/развитие;
• operations – эксплуатация/использование.

Собственно в книге Google и было написано примерно следующее: DevOps - это методология автоматизации технологических процессов сборки, настройки и развёртывания программного обеспечения. И основа этой методологии отнюдь не в конкретных технологиях, а в изменении процесса коммуникации между командами.

6. Почему DevOps?

Закон Конвея. Который вообще-то совсем не закон, а изречение программиста Мелвина Конвея от 1968:

Organizations which design systems (in the broad sense used here) are constrained to produce designs which are copies of the communication structures of these organizations.

Организации проектируют системы, которые копируют структуру коммуникаций в этой организации.

Кстати чуть позже группой исследователей из МТИ и Гарвардской бизнес-школы было опубликовано доказательство в поддержку закона Конвея, можно ознакомиться по ссылке.

Т.е. если мы хотим построить систему, обеспечивающую высокие показатели Time-to-market, то нам нужно от классического подхода с вот такими командами и иерархическим взаимодействием перейти к ... вот этому децентрализованному ужасу.

7. Зачем DevOps?

Внедрение DevOps само по себе не решит автомагическим образом все ваши проблемы. Наоборот, добавит сложности. Однако, иногда эта сложность просто необходима.

В первую очередь это необходимо организациям, которые хотят уменьшить метрику, которые менеджеры называют Time-to-market. То есть когда нужно очень быстро выходить на рынок - минимизация времени от идеи до реализации. И как следствие - для этого необходимы частые выпуски программного обеспечения. И теперь вместо "классических" подходов, когда все стадии жизненного цикла идут одна за другой, все стадии идут одновременно. Отсюда и сложности.

Если условно разделить все команды на два лагеря - Dev (разработчики, тестировщики, аналитики, и др.) и Ops (администраторы и сетевые инженеры), а потом выделить их основные задачи, то получится:

- Dev - Как можно быстрее отправлять изменения и новые фичи на продуктовую среду. Не понимают как оно крутится на серверах.
- Ops - Как можно реже отправлять изменения на продуктовую среду, чтобы было меньше отказов. Не понимают, как работает продукт.
  Буквально противоположные задачи. Вот DevOps должен их подружить и настроить процессы так, чтобы быстро выкатывать и откатывать (что требует автоматизации), при этом ничего не ломать (требует автоматизации тестирования и мониторинг).

8. Цели DevOps.

Выделяют несколько основных целей этой методологии:

- сокращение времени для выхода на рынок;
- снижение частоты отказов новых релизов;
- сокращение времени выполнения исправлений;
- уменьшение количества времени на восстановления при сбое.

9. DevOps или SRE?

Вообще с точки зрения книг Google - DevOps это только методология, которая затрагивает все ранее существовавшие роли - разработчиков, тестировщиков, администраторов. Но с широкой руки или ноги HR-отделов появились отдельные группы DevOps-инженеров. Это очень странный подход, но живем с тем, что имеем.

DevOps-инженер это высококвалифицированный специалист, который отвечает за автоматизацию всех этапов создания приложений и обеспечивает взаимодействие программистов и системных администраторов. Другими словами, DevOps-инженер прорабатывает и автоматизирует сборку и доставку кода от разработки до производства.

Что же такое SRE - Site Reliability Engineering? Это реализация философии DevOps, впервые изобрели и внедрили это в Google в 2003 году. Грань между ними очень тонкая, во всех компаниях роли SR-инженера и DevOps-инженера понимают по-разному, иногда они существуют параллельно. С точки зрения книг Google SRE - это реализация DevOps при использовании разработчиков в качестве системных администраторов и промежуточного звена. В VK - SR-инженер больше про администрирование, сбор метрик, реакции на инциденты. А DevOps-инеженер больше про настройку сборочной цепочки CI/CD.

Специалисты из Google в своих книгах отдельно подчеркивают, что предпочтительнее на должность SRE брать **разработчиков** с большим опытом разработки сложных систем, поскольку они более склонны к автоматизации рутинных действий. Тупому монотонному ручному труду нет места в информационной эре. От него невозможно полностью избавиться, но для максимальной эффективности необходимо стремиться отводить на операционные задачи менее 50% времени, а на инженерные как можно больше. Это позволит сводить к минимуму выгорание сотрудников и затраты на введение нового функционала в систему.

И мне кажется это верно не только для данной методологии. Если вы тратите на поддержку больше времени, чем на развитие, то вы что-то делаете не так, ведь с течением времени количество операционных задач начнет только возрастать, вы похороните себя под ворохом рутины и развитие системы остановится.

В курсе мы больше будем говорить о DevOps, подразумевая конкретную реализацию, описанную в книгах Site Reliability Engineering.

10. Книги.

Подробно SRE описали инженеры из Google в книге Site Reliability Engineering. А затем и в ее продолжении с большей конкретикой - Site Reliability Workbook. Настоятельно рекомендую их к прочтению. Для размышляющих людей это кладезь идей и практик. Что-то слишком идеализировано и оторвано от реальности, но я лично почерпнул очень много полезного. Крайне занимательно и легко читается.

11. Что нужно DevOps/SR-инженеру?

- знать жизненный цикл ПО и методологию
- освоить разные архитектуры ПО, в т.ч. микросервисную архитектуру
- знать основы программирования, выучить несколько ЯП
- уметь отлаживать программы и устранять уязвимости и ошибки
- понимать принципы работы операционных систем
- понимать принципы работы компьютерных сетей
- понимать виртуализацию, облачные и гибридные решения
- разбираться в системах оркестрации
- разбираться в системах управления конфигурацией (СУК)
- уметь налаживать мониторинг продукта
- общаться с другими людьми и командами

Как вы видите, такой инженер это практически универсальная рабочая единица. Как вы должны были уже выяснить из экономики - при капитализме высокий спрос приводит к повышению цены. И действительно это весьма оплачиваемые должности. Из минусов: необходимость постоянно обучаться и знать много из разных областей (такие специалисты совмещают в себе Dev, DBA, SecOps, SysOps, итд).

12. Основные термины.

Система управления исходным кодом (SCCS) — это система контроля версий, предназначенная для отслеживания изменений в исходном коде и других текстовых файлах во время разработки программного обеспечения. Примеры инструментов: Subversion, Git.

Инструменты сборки — это инструменты для управления и организации ваших сборок, которые очень важны в средах, где есть много проектов, особенно если они взаимосвязаны. Они служат для того, чтобы убедиться, что там, где разные люди работают над различными проектами, они ничего не ломают. И чтобы убедиться, что когда вы вносите свои изменения, они тоже ничего не ломают. Примеры инструментов: Maven, Grunt, dpkg.

Системы управления конфигурацией (СУК). Это тесно связано с таким понятием как IaC - Infrastructure as Code, об этом позже. Ручной труд в принципе рано или поздно приводит к ошибкам, так что нам нужно учитывать и это. А автоматизация позволит части этих ошибок избегать. Еще есть рутинная работа, основным рутинным процессом эксплуатации является приведение серверов к определенному состоянию - загрузка ПО, установка пакетов, изменение файлов конфигурации. Это можно и нужно автоматизировать. Для этого есть системы управления конфигурациями. СУК позволяет серверу достичь необходимого заранее определённого состояния (использовать конкретный язык, инструмент или функции). Примеры инструментов: Puppet, Chef, Ansible, Saltstack,

Terraform, Vagrant.

Непрерывная интеграция (англ. Continuous Integration или CI) — это методика загрузки кода в общий репозиторий несколько раз в день с последующей проверкой. Таким образом вы автоматически выявляете проблемы на ранних этапах, когда их проще всего устранить, и развертываете новые возможности для пользователей настолько быстро, насколько это возможно. Примеры инструментов: Jenkins, GitLab-CI, GitHub-CI.

Непрерывная доставка (англ. Continuous Delivery или CD) — это подход к разработке программного обеспечения, при котором программное обеспечение производится короткими итерациями, гарантируя, что ПО является стабильным и может быть передано в эксплуатацию в любое время, а передача его не происходит вручную. Непрерывная доставка отличается от непрерывного развертывания (Continuous Deployment) тем, что процесс развертывания в производственную среду должен быть подтвержден вручную.

Тестирование. Инструменты тестирования охватывают множество потребностей и возможностей, в том числе глубокое тестирование, управление тестированием и оркестрацию. Однако важнейшая функция пакета инструментов DevOps — автоматизация. В долгосрочной перспективе автоматическое тестирование окупается за счет ускорения циклов разработки и тестирования. В среде DevOps оно важно еще и потому, что повышает осведомленность и позволяет раз и на всегда решить какие-то проблемы. Примеры инструментов: Selenium, PyTest, UnitTest, TestNG.

Мониторинг — это постоянный сбор и анализ различных параметров (метрик) поведения системы. С его помощью можно описать и измерить в числовом выражении каждый важный аспект проекта. Данные из разных точек среды собираются системой мониторинга, которая отвечает за хранение, агрегацию, визуализацию данных и автоматические реагирует на изменения, когда значения соответствует заданным условиям. Примеры инструментов: Prometheus, Zabbix, Nagios, Graphite.

Оркестрация — автоматическое размещение, координация и управление сложными компьютерными системами и службами. Контейнерная оркестрация относится к инструментам и платформам, которые используют для автоматизации, управления и планирования приложений, определённых отдельными контейнерами. Инструменты оркестрации контейнеров, как Kubernetes, помогают запускать и управлять всеми вашими контейнерами в производственной среде и решать проблемы, которых могло не быть на стадии разработки на одной машине. Другие примеры инструментов: Docker Swarm.

Облака, облачный хостинг. В отличие от обычного хостинга решения не развертываются на одном сервере. Вместо этого приложение или сайт размещается в сети связанных виртуальных и физических облачных серверов, что гарантирует гибкость и масштабируемость. Примеры инструментов: VK CS, BASIS, Yandex.cloud, Amazon Web Services.

IT-инфраструктура — это комплекс взаимосвязанных компонентов, с помощью которого можно организовать информационную среду и управлять ей. Сюда может относиться как сеть, так и DNS-сервера, серверы репозиториев, и так далее.

13. Основные процессы.

Сборка/тестирование, развертывание и эксплуатация - основные процессы, ради которых все и затевается.

Сборка/тестирование - мы автоматически забираем код из репозитория, формируем некий "пакет" ПО, который затем тестируем и выкатываем на stage/prod окружение.

К развертыванию есть 3 основных требования:

- Наши обновления являются поэтапными. Мы обновляем один экземпляр модуля, проверяем, затем обновляем еще один. И так до тех пор, пока вся система не будет обновлена. Подобным образом, например, работает механизм плавающего обновления (rolling updates) в k8s.
- У нас есть инструменты, которые позволяют быстро выявлять проблемы и помогать при их устранении. Это может быть автоматизированный сбор и анализ логов или метрик. Об этом подробнее поговорим в лекции про мониторинг.
- Мы можем безопасно откатить изменения для дальнейшего разбора полетов и устранения неисправностей. Более того, перед обновлением мы составляем не только план обновления, но и план отката изменений на случай сбоя. Например, предусмотреть возможность вернуться к старой версии схемы базы данных без потери данных за счет переключения на работающую параллельно реплику.

Но нашу систему, да и весь процесс разработки и эксплуатации нужно как-то отслеживать. Как минимум по экономическим соображениям. Для этого SR-инженеры используют:

- Метрики SLA, SLO, SLI. SLA - Service Level Agreement (SLA, соглашение об уровне обслуживания) — это соглашение между вами и клиентами об измеримых показателях, таких как время безотказной работы, время реагирования, а также мерах ответственности. Service Level Objectives (SLO) — это отдельные обещания, которые вы даете клиенту в рамках SLA. Service Level Indicator - Индикатор уровня обслуживания, измеряет соответствие цели уровня обслуживания. Например, если в SLA указано, что системы будут доступны 99,95 % времени, то в качестве SLO, вероятно, будет выбрано время безотказной работы 99,95 %, а в качестве SLI — фактическое измеренное время безотказной работы
- Бюджет на ошибки - интересное нововведение философии. Можно заметить, что 100% надежность системы зачастую избыточна. Разумеется это не касается систем жизнеобеспечения, военных и космических технологий. Но, кажется, что вы не сильно пострадали даже из-за 7-часового отказа Facebook, Instagram и WhatsApp в прошлом году. Что уж говорить о менее значительных заранее запланированных периодах недоступности. Таким образом мы определяем некий допустимый для нас уровень надежности, например 99.5%. Значит у нас есть бюджет ошибок в полпроцента. Кажется мало, но это 7 минут 12 секунд простоя системы в день. А за год почти двое суток. И этот бюджет мы должны тратить так, чтобы добиться максимальной эффективности, например, чаще допускать больший уровень риска при обновлениях функционала. Если же мы этот бюджет значительно не утилизируем, то наша стратегия недостаточно агрессивно позволяет развивать сервис.
- Мониторинг. Мониторинг всего. Каждого модуля, каждой части системы. Максимум данных о системе в каждый момент времени. Об этом также поговорим в отдельной лекции.

А еще мы должны смириться с фактом, что отказы неизбежны и это нормально, нужно планировать свои проекты с учетом сбоев. Аппаратное обеспечение ненадежно, программное обеспечение может содержать ошибки. Диски будут сыпаться, сеть будет моргать, пакеты будут теряться, а излучение будет менять нам биты регистров. И даже пользователи из-за рекламной компании могут "наломиться" на нас все разом. Смирившись с этим печальным фактом, мы придем к:

- Первое. К резервированию. Будем дублировать компоненты до тех пор пока не останется единых точек отказа (SPOF - single point of failure). Несколько коммутаторов, несколько серверов, динамическая маршрутизация, репликация. Об этом поговорим подробнее в лекции про балансировку.
- Второе. К масштабированию, причем преимущественно горизонтальному. Увы у нас нет столько денег, чтобы бесконечно разгонять процессоры и память - упремся в физику. А горизонтальное - это распределение нагрузки между несколькими узлами с помощью балансировщиков. Кстати, зачастую правильный подход к логике приложения позволяет это горизонтальное масштабирование нивелировать, снижая затраты. Отмечу, что архитектура это не только про масштабирование.

> Сделаем небольшое лирическое отступление. Наверное, многие сталкивались с распродажами на Aliexpress. Очевидно это огромный приток пользователей в ограниченный момент времени, что является серьезной проблемой. Как не сложиться под нагрузкой? Можно просто купить миллион серверов, но это дорого. Можно динамически масштабироваться за счет виртуальной инфраструктуры, но это нетривиально. А можно раздавать пользователям купоны, жестко привязанные ко времени. С 10 до 12, с 12 до 14 и так далее. Бинго! Мы только что размазали всех наших пользователей группками на весь день. И теперь это выдержать сильно проще чем всех сразу. Правда платежным системам все равно весь день плохо, но это уже проблемы других SR-инженеров... Вернемся к отказам.

- Третье. Graceful degradation - переводят на русский как "постепенная деградация", мне кажется лучше всего для описания этого подходит термин "кровопускание". При отказе всех компонентов одного типа наша система не должна сразу прекращать свое существование. Или даже хуже - продолжать функционирование со сбоями. Система должна начинать лишаться части функций, это процесс борьбы за живучесть. Мы постепенно отключаем отказавшие подсистемы, но остальные части продолжают работу, это не должно приводить к катастрофе. После возвращения отказавших подсистем функциональность также восстанавливается. Реализовать такой подход можно лишь постоянно держа в голове, что *любые* внешние зависимости могут быть в любой момент временно утеряны. Многие пренебрегают этим, поэтому мы часто видим как перегрузка базы данных вызывает каскадное отключение всех модулей по цепочке зависимости. Система складывается как карточный домик. А стоило просто везде заранее добавить кэширование, timeout-ы и reconnect-ы, а не слепо верить, что соседний сервис будет доступен.

14. Зависимости.

Рано или поздно в долго работающем проекте вообще появятся циклические зависимости и тогда при полномасштабном сбое без соблюдения принципов "graceful degradation" вы станете вести себя как в известном меме. Например, такое случалось у Одноклассников в 2013 году, тогда они вручную перезапускали около пяти тысячи серверов и сервис был недоступен трое суток. Подробнее об этом можно прочитать в их статье на Хабре.

15. Архитектура.

Итак, из всей озвученной философии у нас уже вырисовывается архитектура типичной распределенной отказоустойчивой системы. Горизонтальное масштабирование, резервирование, балансировка нагрузки, мониторинг, пока что ничего необычного.

Согласно нашей философии, для обновлений системы без снижения её доступности нам нужно как минимум N+1 экземпляров каждой части, где N минимальное количество экземпляров, необходимых для обработки нагрузки на систему. Поскольку во время обновления или из-за аппаратных сбоев одна из них может стать недоступна, нам нужно на один больше, думаю понятно. Google при этом советует иметь как минимум N+2, поскольку в момент обновления может произойти аппаратный сбой любого из N компонентов, что приведет к деградации системы, хоть это и маловероятно.

Очень подробно архитектура таких систем рассматривается в дисциплине TFTDS - Theory of Fault-Tolerant Distributed Systems.

16. Инфраструктура как код (IaC)

Подход для управления и описания IT-инфраструктуры через конфигурационные файлы, а не через ручное редактирование конфигураций на серверах или интерактивное взаимодействие. Этот подход может включать в себя как декларативный способ описания инфраструктуры, так и императивный.

17. Где размещать проект?

Имеем следующие варианты:

- На собственных серверах
- На виртуальных машинах в облаке
- В Platform as a Service / Serverless

18. Ретроспектива развития виртуализации

Проведем ретроспективу.

- Сервер с одним сервисом
- Сервер с набором сервисов
- Сервер с набором виртуальных машин
- Виртуальные машины в облаке
- Контейнерная виртуализация
- Оркестрация

- PaaS / Serverless

Сначала мы просто потребляли один сервер одним приложением и вариантов не было, но потом севера стали наращивать мощность.

Рассмотрим минусы в случае с отдельным сервером. Сервера как правило толстые - там много ядер, много памяти (иначе просто экономически не выгодно). И получается, что наш сервис потребляет только маленькую часть ресурсов этого сервера и вне пиковых нагрузок железо просто простаивает - греет воздух, бизнес теряет деньги. Более того, нам потребуется самим настроить железо - сам сервер, сеть, и так далее.

Для устранения несправедливости по части простоя было решено размещать несколько сервисов на один сервер. Тут же появилась серьезная проблема изоляции - один сервис мог помешать другому. Сначала это решали на уровне фс - отделяли правами, точками монтирования и chroot.

Затем появилась проблема, что наши сервисы могут подраться за один порт, появились Namespaces.
Сразу следом возникает вопрос - как заставить приоритизировать или ограничить часть ресурсов для одного сервиса? Появились Cgroups.

Параллельно с этим подходом развился подход виртуализации - на одном сервере запускаем множество разных операционных систем, делим между ними ресурсы. Причем запускать любые ОС. Сразу же появилось множество сервисов, которые предоставляли за плату на своих серверах ВМ, это назвали облаками. Лучший пример - AWS, стремительный взлет на рынке.

Но начали развиваться микросервисные архитектуры и мы снова получили массу сервисов внутри одной виртуальной машины. Попытка решить это разнесением на несколько виртуальных машин привела к проблемам производительности, начали сказываться накладные расходы на запуск отдельного ядра ОС.

Вот тут на арену и вышло закономерное развитие прошлого подхода с пространствами имен и ограничением ресурсов - появилась контейнерная виртуализация или контейнеризация. Контейнер не эмулирует работу виртуальных устройств и не запускает отдельное ядро ОС, все команды проходят через ядро хостовой ОС. Таким образом накладные расходы на запуск контейнера практически эквивалентны запуску процесса, с поправкой на создание отдельных пространств имен.

Управление массой контейнеров тоже в какой-то момент стало проблематичным. Тогда начинают появляться первые оркестраторы - Docker Swarm, k8s и другие. Они позволяют удобно управлять целым массивом контейнеров, что значительно упрощает работу с большими сложными проектами.

Но и на этом сообщество не остановилось. Некоторые заметили, что во-первых есть масса похожих задач, например балансировка нагрузки или поднятие СУБД и это можно превратить в сервис, без необходимости нам самим заниматься настройкой. Наше приложение просто использует готовые инстансы, а провайдер занимается настройкой и управлением. Так появились PaaS-сервисы. Во-вторых, часть задач просто не нужно выполнять постоянно, например резервное копирование или обработка запросов. Отсюда возникли сервисы, которые выполняют наш код только тогда, когда он требуется. У нас нет

постоянно работающего процесса, виртуалки или контейнера, они появляются по мере необходимости, поэтому такой подход назвали serverless, мы как бы вообще не контактируем непосредственно с низким уровнем.

19. Виртуализация

Виртуализация — предоставление набора вычислительных ресурсов или их логического объединения, абстрагированное от аппаратной реализации, и обеспечивающее при этом логическую изоляцию друг от друга вычислительных процессов, выполняемых на одном физическом ресурсе.

Гипервизор - программа или аппаратная схема, обеспечивающая или позволяющая одновременное, параллельное выполнение нескольких операционных систем на одном и том же хост-компьютере.

20. Классификация виртуализации

- Эмуляция - полная виртуализация аппаратной платформы, например QEMU.
- Программная виртуализация
    - Трансляция команд - перехват и переделка команд на стороне гипервизора
    - Паравиртуализация - модификация ядра ОС для работы с API гипервизора
- Аппаратная виртуализация - виртуализация с поддержкой специальной процессорной архитектуры - требует поддержки со стороны аппаратного обеспечения (Intel VT / VTd / VMX или AMD-V / SVM)
- Контейнеризация - виртуализация на уровне операционной системы, работа нескольких экземпляров пространства пользователя в рамках одной ОС. Подробнее об этом поговорим во второй лекции.

21. Гипервизоры

Задачи гипервизора:

- эмуляция виртуальных аппаратных ресурсов
- полная изоляция среды
- распределение физических аппаратных ресурсов

Гипервизоры 2 типа:
Устанавливаются в существующей ОС. Это делает его размещенным гипервизором, поскольку он использует ОС хост-компьютера для выполнения определенных операций, таких как управление вызовами к ЦП, управление сетевыми ресурсами, управление памятью и хранилищем. Это позволяет гипервизорам типа 2 поддерживать широкий спектр аппаратного обеспечения.
Примеры: VirtualBox, VMware Workstation, QEMU, Parallels, Microsoft Virtual PC.

Гипервизоры 1 типа:
Работают непосредственно на аппаратном оборудовании хост-машины - не нужно загружать базовую ОС для его работы. Т.е. гипервизоры типа 1 имеют прямой доступ к оборудованию, что делает этот тип гипервизора наиболее эффективным и производительным.
Примеры: VMware ESXi, Citrix XenServer.

Иногда выделяю гибридные гипервизоры, которые могут совмещать подходы обоих типов.

Одной из интересных технологий является гипервизор KVM. Этот гипервизор на основе Linux с открытым исходным кодом классифицируется как гипервизор 1 типа, который превращает ядро Linux в «железный» гипервизор, что позволяет работать KVM-виртуализации параллельно ядру Linux. Поверх KVM работает масса других проектов - oVirt, Proxmox, OpenStack и так далее.

22. Облачные решения

Amazon Web Services
Google Cloud Platform
Microsoft Azure
IBM cloud computing
VK CS (MCS)
Yandex.cloud
BASIS

> Снимки экрана

23. GNU/Linux

GNU/Linux - семейство операционных систем на основе ядра Linux и программ проекта GNU. Не являются системами семейства Unix, однако работают по схожим принципам, частично соответствуют стандартам POSIX и признаются Unix-подобными.

24. Командная оболочка

Командная оболочка Unix (англ. Unix shell, часто просто «шелл» или «sh») — командный интерпретатор, используемый в операционных системах семейства Unix, в котором пользователь может либо давать команды операционной системе по отдельности, либо запускать скрипты, состоящие из списка команд.

Bash (от англ. Bourne again shell, каламбур «Born again» shell — «возрождённый» shell) — усовершенствованная и модернизированная вариация командной оболочки Bourne shell. Одна из наиболее популярных современных разновидностей командной оболочки UNIX. Особенно популярна в среде Linux, где она часто используется в качестве предустановленной командной оболочки.

```
$# – общее количество параметров переданных скрипту
$* – все аргументы переданные скрипту(выводятся в строку)
$@ – тоже самое, что и предыдущий, но параметры выводятся в столбик
$! – PID последнего запущенного в фоне процесса
$$ – PID самого скрипта
```

```
#!/bin/bash

var1=$1 – первый параметр скрипта
var2=$2 – второй параметр скрипта

if [[ "$var" -eq "substring" ]]
then
  echo "Равно"
else
  echo "Не равно"
fi

for i in {1..10}
do
  echo "Номер $i"
  echo 'Номер $i' # Просто текст
done
```

Статьи для новичков Основы BASH и Как писать bash-скрипты надежно и безопасно: минимальный шаблон.

25. Chroot

Операция изменения корневого каталога в Unix-подобных операционных системах. Тут речь либо про системный вызов, либо про утилиту, которая его делает. Самая простейшая изоляция, не является безопасной, т.к. процесс с правами root все еще может сделать повторный вызов chroot.

26. Namespaces

Namespace - механизм изоляции и группировки структур данных ядра. Внутри процессы нумеруются с 1, как на пустом сервере. Другие процессы вне пространства имен обнаружить нельзя. Посмотреть можно в /proc/$PID/ns.

В ядре Linux существует структура task_struct, которая описывает процесс.
Пробежимся по основным параметрам:
pid_t pid;
pid_t tgid;
nsproxy *nsproxy;

Namespaces syscalls:

- clone() - склонировать процесс, при создании процесса можно выбрать флаги, которые будут показывать какие пространства имен нужно передать или создать
- unshare() - можно создать себе индивидуальное пространство имен
- setns() - установить определенное пространство имен

Namespaces
mount - пространство фс - копия дерева файловой системы, ассоциированная с процессом
uts - пространство имени хоста и доменного имени
ipc - пространство ресурсов межпроцессного взаимодействия
pid - пространство номеров процессов, потомок внутри с pid 1 имеет родителя 0, т.е. невозможно понять иерархию снизу вверх, но можно сверху вниз.
network - пространство имен сетевых настроек (интерфейсов, маршрутизации) - управляется через `ip netns ...`
user - пространство номеров пользователей - можно заставить думать процесс, что он запущен от root.

Чтобы создать пространство имен, которое не умрет при завершении потомка, мы можем воспользоваться `mount --bind` на /proc/$PID/ns.

27. CGroups

Control groups - механизм изоляции ресурсов ядра. Работает поверх sysfs. Описывают иерархию ресурсов.

sysfs - псевдофайловая система, позволяет создать иерархию объектной модели на файловой системе. Каталог - объект, файл внутри - атрибут. Можно читать/писать атрибуты, атрибуты могут быть составными.

Посмотрим список подсистем, которые могут ограничивать что-то.

```
ls /sys/fs/cgroup/
```

blkio — устанавливает лимиты на чтение и запись с блочных устройств;
cpuacct — генерирует отчёты об использовании ресурсов процессора;
cpu — обеспечивает доступ процессов в рамках контрольной группы к CPU;
cpuset — распределяет задачи в рамках контрольной группы между процессорными ядрами;
devices — разрешает или блокирует доступ к устройствам;
freezer — приостанавливает и возобновляет выполнение задач в рамках контрольной группы
hugetlb — активирует поддержку больших страниц памяти для контрольных групп;
memory — управляет выделением памяти для групп процессов;
net_cls — помечает сетевые пакеты специальным тэгом, что позволяет идентифицировать пакеты, порождаемые определённой задачей в рамках контрольной группы;
netprio — используется для динамической установки приоритетов по трафику;
pids — используется для ограничения количества процессов в рамках контрольной группы.

Создать новую группу в подсистеме cpuset

```
mkdir /sys/fs/cgroup/cpuset/group0
```

Добавим процесс нашего текущей командной оболочки в группу, смотрим доступные процессу ядра

```
echo $$ > /sys/fs/cgroup/cpuset/group0/tasks
cat /proc/$$/status | grep '_allowed'
```

Привяжем процессы группы к 0-му ядру, смотрим доступные процессу ядра

```
echo 0 >/sys/fs/cgroup/cpuset/group0/cpuset.cpus
cat /proc/$$/status | grep '_allowed'
```

Теперь создадим новую группу в подсистеме memory, добавим туда наш шелл и ограничим 40 MiB

```
mkdir /sys/fs/cgroup/memory/group0
echo $$ > /sys/fs/cgroup/memory/group0/tasks
echo 40M > /sys/fs/cgroup/memory/group0/memory.limit_in_bytes
```

Про более новые cgroup v2 можно почитать в статье и руководстве.