# DevOps

ДЗ: https://github.com/netology-code/sysadm-homeworks

Главная задача DevOps инженера — максимально увеличить предсказуемость, эффективность и безопасность разработки ПО.

Devops – работа на стыке нескольких должностей:

Software Engineering (Разработка)-Quality Assurance (Тестирование)-Technology Operations (Эксплуатация)

Также иногда в это понятие добавляют и ответственность за безопасность: Security (Безопасность)

В итоге специалист DevOps имеет достаточно большую ответственность: DevSecOps

## Разработка

Особенности шага:

●Все разработчики используют идентичную среду разработки

●Разработчики должны писать тесты (юнит, функциональные, приемочные)

●Необходимо иметь возможность проводить код-ревью и быстро получать результаты сложных тестов

## Тестирование

Особенности шага:

●Тестирование должно быть автоматизировано

●В основную ветку не должен попасть нерабочий код

●Новый код должен быть покрыт тестами

## Безопасность

Особенности шага:

●Наличие контроля доступа для разных сотрудников

●Исключение известных уязвимостей в используемых библиотеках

●Наличие контроля доступа отдельных сервисов

## Создание песочницы, выкладка на стейдж

Особенности шага:

●Каждая фича должна быть протестирована на изолированном окружении

●Все окружения должны быть идентичны

●Процесс создания и уничтожения песочниц должен быть автоматизирован

## Выкладка в продакшн

Особенности шага:

●Выкладка в продакшн должна быть только после успешного прохождения тестов ●Выкладка и отклад изменений должны происходить автоматически

●Должна быть возможность откатки изменений

## Мониторинг

Особенности шага:

●Все ошибки во время тестирования на стейдже и при работе приложения в продакшене должны стать известны

●Должен быть простой доступ к логам всех составляющих системы

●Должно быть автоматическое оповещение об ошибках

# Настройка рабочей среды

## Выбор общих инструментов

Зачем использовать общие инструменты?

●Будет проще понимать друг друга

●Подсветка синтаксиса

●Автодополнение кода и команд

## IDE PyCharm Community Edition

Почему IDE PyCharm?

●Бесплатная редакция

●Доступны все необходимые плагины

●На базе самого популярного решения от IDEA

## Консоль в разных ОС

Выбор терминала / консоли:

●MacOS. Iterm. <https://www.iterm2.com/>

●Windows. WSL: <https://docs.microsoft.com/ru-ru/windows/wsl/install-win10>

●Linux. Стандартный или ваш любимый терминал

# Этапы разработки ПО

## Коммуникация

Задача этапа - отлаженное взаимодействие:

●Между заказчиком и менеджером

●Между менеджерами

●Между менеджером и разработчиками

●Между разработчиками

●И др.

## Разработка

Задачи этапа:

●Выбор инструментов

●Написание кода

●Внутреннее тестирование

●И др.

## Внешние интеграции

Общие задачи этапа:

●Лицензирование

●Соглашение о модели взаимодействия

●Поддержка изменений

●И др.

## Инфраструктура

Задача этапа – создать отказоустойчивое решение:

●Выбор оптимального решения

●Развертывание нескольких площадок

●Дальнейшая поддержка

●И др.

## Тестирование

Задача этапа - прохождение тестирований:

●Юнит тестирование

●Функциональное тестирование

●Интеграционное тестирование

●И др.

# Подходы к разработке.

## Agile

Agile - гибкая методология, которая позволяет вашему проекту меняться легко и быстро. Обычно включает себя практики Scrum и Kanban. Работает по принципу: требования > план > работа > ревью > повтор.

Как работать?

●оцените этапы работы по часам (часто оценка является приблизительной)

●установите приоритетность задач (самые важные — в начало очереди)

●проанализируйте уже выполненную работу (если вы не вписываетесь в план, увеличьте рабочую нагрузку над этим спринтом)

## Agile манифест

Главное в Agile манифесте:

●Люди и взаимодействие важнее процессов и инструментов

●Работающий продукт важнее исчерпывающей документации

●Сотрудничество с заказчиком важнее согласования условий контракта

●Готовность к изменениям важнее следования первоначальному плану

## Основные фреймворки

Фреймворки, которые включает в себя методология Agile:

●Scrum

●Kanban

Пример организации процесса

●Деление работы на части - спринты (1-2 недели)

●Спринты планируются исходя из требований для этого конкретного момента времени

●Относительная оценка времени выполнения работ

●Ревью каждого спринта, чтобы понять, как он прошёл и что можно было бы улучшить

●Фидбек по поставляемому продукту

●Ежедневные собрания (очень короткие)

●Еженедельные собрания

●Непрерывная разработка

●Визуализация процесса на доске

●Решение сначала самых важных задач

●Поэтапные улучшения

# Работа в Терминале

## Общие цели модуля

●Научиться работать в командной строке Linux – на серверах графической оболочки не будет :)

●Понять, как подходить к возникающим проблемам.

●Углубить знания о ресурсах операционной системы Linux и оценке их потребления.

●Узнать, понимать сетевые протоколы разных уровней, получить навыки взаимодействия с ними на уровне администратора.

## Для чего основы администрирования полезны DevOps в теории...

●Подготовиться к самостоятельному развертыванию сервисов при необходимости. Нередко функциональности облачных managed-сервисов не хватает, или из-за особенностей ПО их нельзя использовать в нужном режиме.Пример: в GCP до сих пор нет MySQL 8, Huawei Cloud застрял на Kubernetes годовалой давности без cri-containerd (недоступен gVisor и т.п.)

●Научиться экономить денежные средства при самостоятельном развертывании по сравнению с managed-сервисами. Часто с разумными затратами усилий можно сделать инфраструктуру существенно дешевле и не хуже (а может быть – даже и лучше, см. SLA на переключение в отказоустойчивых Google Cloud SQLHA инстансах), чем предлагают облачные провайдеры.

●Научиться поддерживать самостоятельно развернутые сервисы. Очень часть DevOps – про облака. Однако много компаний используют собственное железо или имеют гибридную инфраструктуру, а значит спрятаться только лишь за абстракции terraform'а не получится.

## и на практике

●Научиться разбираться с ПО, которое работает не так, как заявлено в документации.

●Научиться управляться с внутренними разработками. Авторы давно уволились, времени/знаний разбираться в исходниках нет, надо срочно починить, на stackoverflow ответов по понятным причинам нет.

●Научиться строить системы с пониманием компромиссов. Часто необходимо создать оптимизированную по заданным параметрам архитектуру: отказоустойчивость/стоимость/производительность. Не бывает все и сразу.

●Научиться оценивать производительность и влиять на нее с пониманием. Разобраться в недостаточной производительности в продакшене того, что работает нормально в других средах, а не просто “увеличить инстанс в Амазоне”.

# Немного о структуре директорий Linux

## FHS (filesystem hierarchy standard).

Некоторые характеристики:

●единый корень всей файловой системы , ‘/’, forward slash

●все директории в дереве внутри относительно корня

●если вы из мира Windows – большое отличие в том, что разделы также находятся внутри относительно единого корня

●стандарт Filesystem Hierarchy Standard○/bin, /sbin, /etc...

●многие директория называются понятно:○/home для домашних каталогов пользователей○/root для домашнего каталога аккаунта администратора○/tmp для временных, temporary, файлов

Что мы видим при открытии терминала?

Один из командных интерпретаторов:

●sh, bash, zsh, fish, и т.д.

Стартует сессия, элементы которой мы сразу наблюдаем:

●имя пользователя

●имя хоста

●тильда – сокращенное представление домашнего каталога, home directory

●приглашение ко вводу

Как еще посмотреть имя пользователя?

**$whoami**

## В каком формате shell ожидает ввод?

Базовая структура интерактивного взаимодействия – команда, аргументы этой команды. Начинаем знакомиться с некоторыми командами и действиями в терминале Linux:

●echo Hello world

●echo $USER

●$SHELL –-version

●new\_variable=new\_text

●read

●history

●clear, reset (aafire)

●управление с клавиатуры: Сtrl + L, Up/Down, Ctrl + R

●автодополнение по Tab: bash autocompletion (‘\ ‘)

●# и пробел

●env – переменные окружения (environment)

Страница

## Команды просмотра директорийи переходов между ними

●pwd – текущая директория (present working directory)

●cd – изменить директорию (change directory)

○. / ..

○-

●ls – показать содержимое (list)

○-l

○-a

○-la, сочетание опций

●скрытые объекты с точкой

## Alias

Удобный инструмент для сокращения длины вводимых команд – alias.

●ll, алиас для ls -alF

●alias my\_new\_shiny\_alis="ll -t"

Посмотреть список всех присутствующих alias,

●alias без параметров

Не только сокращения: один из способов подмены команд (alias docker='podman')

Как сделать изменения алиасов постоянными (персистентными)?

●~/.bashrc

●редактирование файлов в nano

●символ ^ как Ctr

## Команды для просмотров файлов

... и поиска содержимого в них.

Простейший просмотр содержимого:

●cat – conCATenate

●head / tail – начало и конец файла

○-n

○<число>, -/+<число>

Просмотр содержимого в режиме пейджера (постранично):

●less vs. More

○-i

○-N

Некоторые ключи grep для фильтрации (поиска) содержимого файлов:

○-c

○-i

○-v

○-f (diff, diff -u + sha256sum)

## Команды для создания, перемещения и удаления директорий и файлов

●mkdir – make directory

○-p

●touch

●cp – copy

○-v

○-r

○\* (strace git add \*)

●mv – move

●rm – remove

○-d

○-r

Страница

# Встроенные функции, ключевые слова и программы

Почему не просто команды

●type echo: echo is a shell builtin

●type whoami: whoami is /usr/bin/whoami

●type {: { is a shell keyword

type type :)

type -a pwd

Разные shell – разные команды.

Как shell понимает, что вы от него можете хотеть? Что за /usr/bin/whoami?

●echo $PATH; export$PATH ($GREP\_OPTIONS)

●which – порядок в $PATH

●env pwd

●compgen -k, compgen -b

Документация по bash – 4000 строк по состоянию на апрель 2020 (огромное количество встроенных ключевых слов и возможностей, которые стоит изучать).

Страница

из

Документация, как искать сведения о командах

Не просто google it!

●разные среды исполнения и дистрибутивы

●разные версии программ даже в рамках одного дистрибутива

●саморазвитие

Различные варианты поиска по встроенной документации

●man -k <keyword>

●man -P <pager>

●секции man – открывает первую найденную, man printf = man 1 printf, man 3 printf

●-h / --help

●info

●help

Один из базовых пакетов с набором обсуждаемых в данной лекции программ – coreutils, или “базовые” утилиты. Их невозможно рассмотреть все в рамках лекций, так как состав пакета обширный: <https://ru.wikipedia.org/wiki/GNU_Coreutils>

Как минимум стоит знать о составе пакета и почитать help на команды, которыми вы пользуетесь ежедневно.

## Средства управления задачами bash

По умолчанию процессы запускаются в активном режиме, “на переднем плане”.

●Сtrl + С: прервать

●Ctrl + D: завершить

●Ctrl + Z: отправить в фон, background, bg

●Ctrl + S, Ctrl + Q: приостановить и продолжить

Управление фоновыми задачами (“заднего плана”):

●jobs: посмотреть список

●fg %<id>

Планирование выполнения задач в будущем:

●at / atq / atrm

## Изучайте инструменты, которые используете

●Размеры/шрифты/темы

●$PS1

●Базовое поведение можно менять – stty -ixon и Ctrl + S

●Горячие клавиши: Esc + ., !!

●Emacs-режим vs vi-режим

●shopts (dotglob)

# Стандартные потоки ввода-вывода: stdin, stdout, stderr

## Как shell понимает, откуда ждать ввод и куда направлять вывод?

На прошлой лекции мы познакомились с несколькими командами, которые предполагают ввод...

$read new\_varHello world!

... и вывод данных:

~$ echo $new\_varHello world!

Когда вы работаете в терминале за локальным компьютером, может показаться само собой разумеющимся, что ввод и вывод в сессии происходит с физически подключенных устройств (ваша клавиатура, экран вашего компьютера).

Но всегда ли это так, и как именно оболочка принимает решение об обращении с потоками данных?

Потоки данных – одна из абстракций ОС

Использовать в программах прямой доступ к устройствам через драйвер – не универсально. Например, у удаленного сервера нет физической клавиатуры, но он как-то должен получить команды, переданные с вашей клавиатуры.

●stdin или standard input – стандартный ввод, условный номер 0

○ввод команды и ее аргументов в shell: ls -l, man ls

●stdout или standard output – стандартный вывод, условный номер 1

○вывод результата работы команды: листинг директорий ls, документация man

●stderr или standard error – стандартный вывод ошибок, условный номер 2

○вывод ошибок при работе команды: сообщение о вызове ls на несуществующую директорию, man на отсутствующую команду

Потоки могут не содержать данных:

●в случае успеха $**cp** без дополнительных опций не сообщит об успешном копировании,

●не о чем будет сообщать и в потоке ошибок.

# Процессы

Процесс – экземпляр запущенной программы.

Находясь в bash и вызывая внешние по отношению к оболочке программы (вспоминая первую лекцию – не builtin или не keyword, они как раз выполнятся внутри bash), мы порождаем новые процессы.

Хотя многие простые команды могут отработать и завершиться быстро, во время своего существования эти короткоживущие процессы также полноправны в системе как и, скажем, сам bash, или долгие ”тяжелые” процессы вроде виртуальных машин Java

## Нумерация файловых дескрипторов

Цифры 0, 1 и 2 называются файловыми дескрипторами (file descriptors, FD) и в общем случае стандартный процесс Linux содержит эти потоки данных по умолчанию.

Как увидеть данные потоки, если известны лишь их абстрактные номера?

Здесь нам поможет **lsof** (list open files), или список открытых файлов.

## Отношение файлов, файловых дескрипторов и стандартных потоков

Вы спросите, причем здесь список открытых файлов, и почему вдруг файловые дескрипторы, если мы хотим посмотреть на стандартные потоки процесса?

Нередко встречается выражение “все в Linux является файлом”, и в данном случае эта фраза поможет ответить на заданный вопрос: у потоков std{in,out,err}существует виртуальное файловое представление.

Открытие любых файлов приводит к выделению им файловых дескрипторов, все они получают номера.

## Псевдо-файловые системы /dev и /proc

Мы уже рассматривали операции над реальными файлами, находящимися на существующей файловой системе: научились их просматривать cat или less, перемещать mv, удалять rm и т.д.

Кроме файловых систем, предназначенных для работы с физическими накопителями, в Linux существуют и виртуальные псевдо-файловые системы, две из которых мы сегодня затронем:

●/dev (файловое представление устройств)

●/proc (файловое представление структур ядра)

## Read-only действия с псевдо-файловыми системами

Базовые операции с виртуальными служебными файловыми системами не отличаются от реальных файлов и директорий:

~$ ls /proc/up\*

/proc/uptime

~$ cat /proc/ uptime # время со включения в секундах, время в idle

2465.32 1911.94

Каталог /proc содержит множество сервисной информации, которая является актуальной на момент просмотра и постоянно изменяется.

Чтение из /dev на примере клавиатуры

Среди прочего в /dev представлены и физические устройства, такие как мышь и клавиатура. Это “виртуальные” файлы, представление которых реализуют драйверы. Почитаем из устройства клавиатуры:

evtest /dev/input/event2

Этот выходной поток данных может являться входным для ожидающей ввода программы.

## Атрибуты процессов: PID

Чтобы воспользоваться lsof и исследовать потоки данных, нам необходимо знать базовые атрибуты процесса – ведь к нему, как минимум, надо как-то обратиться.

PID, Process IDentifier - числовой идентификатор процесса, однозначно его определяющий. Находясь в shell, мы можем узнать собственный PID через зарезервированную переменную

$:echo $$

Для задач: jobs -l

Поиск по Special Parameters в man bash расскажет о других подобных переменных.

## Атрибуты процессов: PPID

PPID, Parent PID - атрибут процесса, определяющий идентификатор его родительского процесса. Все процессы в Linux выстроены в дерево и должны иметь “родителя”. Так, для sleep 1h родителем будет bash, из которого sleep вызвали.

Для PPID в shell так же есть одноименная переменная: echo $PPID

Воспользуйтесь pstree -p, если хотите посмотреть графическое представление всех существующих в ОС процессов в виде дерева.

Часть информации в /proc представлена с группировкой по PID: в /proc/<PID>находятся относящиеся к конкретному процессу данные.

## lsof -p $$

Теперь мы обладаем достаточными знаниями, чтобы выполнить данную команду и понять ее вывод

Запросим список открытых файлов для процесса с PID нашего shell:

~$ lsof -p $$

COMMAND PID USER FD TYPE DEVICE SIZE/OFF NODE NAME

...

bash 2020 rgersh 0u CHR 136,0 0t0 3 /dev/pts/0

bash 2020 rgersh 1u CHR 136,0 0t0 3 /dev/pts/0

bash 2020 rgersh 2u CHR 136,0 0t0 3 /dev/pts/0

Альтернативно:

~$ ls -l /proc/$$/fd/{0,1,2}

lrwx------ 1 rgersh rgersh 64 21:08 /proc/2020/fd/0 -> /dev/pts/0

lrwx------ 1 rgersh rgersh 64 21:08 /proc/2020/fd/1 -> /dev/pts/0

lrwx------ 1 rgersh rgersh 64 21:08 /proc/2020/fd/2 -> /dev/pts/0

Так как же shell понимает, откуда ждать ввод и куда направлять вывод?

Таким образом, ответ на изначальный вопрос раздела:

shell и его потомки понимают, откуда ждать ввод и куда направлять вывод, благодаря стандартным потокам, которые в нашем случае указывают на некий **/dev/pts/X**

## Стандартные потоки – атрибут процесса

Представление /dev четко демонстрирует, что стандартные потоки являются атрибутом процесса. Используя /dev/stderr, можно сослаться только на свой собственный поток ошибок, /dev/stderr будет “свой” у каждого процесса

~$ ls -l /dev/std\* # self – ссылка на “самого себя”

lrwxrwxrwx 1 root root 15 21:07 /dev/stderr -> /proc/self/fd/2

lrwxrwxrwx 1 root root 15 21:07 /dev/stdin -> /proc/self/fd/0

lrwxrwxrwx 1 root root 15 21:07 /dev/stdout -> /proc/self/fd/1

Зато эти потоки можно перенаправлять:

~$ bash 2>/dev/nulllsof -p $$

COMMAND PID USER FD TYPE DEVICE SIZE/OFF NODE NAME

...

bash 2568 rgersh 0u CHR 136,0 0t0 3 /dev/pts/0

bash 2568 rgersh 1u CHR 136,0 0t0 3 /dev/pts/0

bash 2568 rgersh 2w CHR 1,3 0t0 6 /dev/null

## > – перенаправление выходных потоков

Без id подразумевает stdout, т.е. echo $$ > /tmp/bash\_pid эквивалентно echo $$ 1> /tmp/bash\_pid

Частый пример использования: >/tmp/log 2>&1, что означает:

●перенаправить stdout в файл /tmp/log,

●перенаправить stderr в stdout,

то есть собрать в /tmp/log одновременно и stdout, и stderr.

>> – режим открытия файла.

Почему не будет работать 2>&1 >/tmp/log?

Почему не будет работать sed 's/foo/bar/' file >file?

Порядок важен!

1.Если сначала перенаправить stderr, он будет указывать на текущее значение stdout (/dev/pts/X), то есть для stderr ничего не изменится.

2.Все перенаправление устанавливается shell до начала выполнения программы

## < – перенаправление входного потока

Перенаправление входного потока подразумевает **stdin**

~$ echo DevOps@Netology > learning

~$ read new\_var < learning

~$ echo $new\_varDevOps@Netology

Если попытаться выполнить некорректную операцию, например, записать что-то в stdout, ОС не даст этого сделать:

~$ ls 1< /tmp/1

ls: write error: Bad file descriptor

## | – pipe

Позволяет связать stdout одного процесса с stdin другого:

ls -R /usr | less.

Можно делать цепочки из нескольких pipe.

Не применяйте pipe без необходимости. Часто встречаются конструкции вида cat some\_file | grep some\_string, тогда как можно обойтись одиночной grepsome\_string some\_file.

Наследование файловых дескрипторов

При создании новых процессов в Linux дочерний процесс автоматически получает от родительского определенный набор свойств. В этот список входят и открытые файловые дескрипторы. Именно поэтому процессы, которые мы создаем из bash, по-умолчанию, используют те же самые стандартные потоки, что и оболочка:

~$ sleep 1h & ls -l /proc/$!/fd

[1] 4859

total 0

lrwx------ 1 rgersh rgersh 64 апр 26 23:25 0 -> /dev/pts/0

lrwx------ 1 rgersh rgersh 64 апр 26 23:25 1 -> /dev/pts/0

lrwx------ 1 rgersh rgersh 64 апр 26 23:25 2 -> /dev/pts/0

Перенаправим вручную stdout на соседнюю вкладку:

~$ echo Follow the white rabbit >/dev/pts/1

Но что же все-таки это за устройство /dev/pts/0, и кто запустил bash со связанными с ним потоками?

# Терминал TTY, эмулятор терминала TTY, псевдо-терминал PTY

## Терминал TTY

Чтобы понять, что за устройство /dev/pts/X, нужно обратиться к истории. Прародитель Linux (\*nix, о котором будем говорить в следующей лекции) был изначально многопользовательской ОС, предполагавшей подключение множества устройств ввода-вывода к компьютеру.

О графическом интерфейсе тогда речи не шло, был доступен только текст, видеокарты в компьютере не существовало.

Устройство для передачи текстовых символов, подключающееся по последовательному порту к компьютеру через UART (Universal Asynchronous Receiver and Transmitter), – это и есть аппаратный терминал, teletypewriter или TTY

Данные с UART обрабатывались на уровне ядра операционной системы в модуле line discipline и драйвере TTY.

Они реализовывали:

●буферизацию вводимых символов,

●реакцию на специальные символы, такие как перевод строки/backspace или комбинации вроде Ctrl + C,

●вывод через UART переданных данных назад на TTY, чтобы пользователь сразу мог видеть вводимые им символы.

При нажатии Enter, буферизованные символы сбрасывались в stdin программе, которая выполнялась в этой сессии и находилась в активном режиме (foreground, вспомните bg/fg и jobs из предыдущей лекции). Много процессов, но активное взаимодействие только с одним.

Например, при начале работы с системой этой программой могла быть login, необходимая для аутентификации пользователя. Затем, как вы уже догадались, ей становился shell.

## Эмулятор терминала TTY

С развитием техники и физическим уменьшением компьютеров необходимость в аппаратных терминалах отпала, однако до появления современных интерфейсов оставалось еще много времени.

Следующим этапом в развитии стал эмулятор терминала, то есть программа, которая позволила пользоваться привычным окружением на оборудовании, в котором аппаратные терминалы уже отсутствовали.

Прямой доступ к эмулятору терминала в Linux имеется до сих пор: нажатие Ctrl + Alt + F1..F6 позволяет запустить несколько независимых сессий, как будто к вашему компьютеру подключено несколько аппаратных терминалов.

Возврат в графический режим осуществляется через Ctrl + Alt + F2

В примере stdin, stdout и stderr shell будут установлены в /dev/ttyX вместо /dev/pts/X в этом режиме, так как используется драйвер TTY, а не PTY.

## Псевдо-терминал PTY

Важной особенностью эмулятора TTY является то, что он реализован в ядре ОС, а значит должен быть максимально универсален, не требователен к ресурсам и т.д.

В современной реальности, когда доступно несравнимо больше аппаратных ресурсов, а в графическом интерфейсе пользователи хотят удобства, было принято решение для этих целей не расширять имеющийся эмулятор, а создать новую концепцию псевдо-терминала (pseudo-terminal, PTY).

PTY – пара виртуальных устройств:

●/dev/ptmx для PTY-master

●/dev/pts/X для PTY-slave

Сами драйверы этих виртуальных устройств, как и драйвер TTY, работают на уровне ядра, однако если раньше и сам эмулятор терминала был частью ядра, теперь это пользовательское приложение, лишь взаимодействующее с мастером PTY.

Фактически, PTY-slave продолжает выполнять те же функции, которые выполнял ранее драйвер TTY, взаимодействуя с модулем line discipline. Изменилась только схема доставки данных в и из него. Если раньше они поставлялись от аппаратного терминала или от ядерного модуля эмулятора терминала, теперь ядро занимается доставкой данных между PTY slave и master.

При открытии эмулятора в графическом интерфейсе происходит следующее:

1.приложение делает вызовы к X Window (X11) для отрисовки интерфейса, визуально схожего с оригинальным терминалом, но не ограниченного ресурсами ядра с любыми эффектами вроде цветов, прозрачности, тем, прокруткой, поддержкой системного буфера обмена и т.д.

2.начинает “слушать” системные события от X Window, такие как нажатия клавиатуры

3.делает специальный вызов к /dev/ptmx, создавая парную сессию /dev/pts/X

4.события отправляются в мастер /dev/ptmx, где вступает в дело line discipline

5.полученные из /dev/ptmx в ответ данные выводятся в графический интерфейс

То есть, само приложение терминала работает именно с PTY-master:

~$ lsof -p $(pgrep gnome-terminal) | grep /dev/pt

gnome-ter 6802 rgersh 20u CHR 5,2 0t0 88 /dev/ptmx

Страница

и Далее эмулятор терминала запускает дочерний процесс – shell того пользователя, под которым произведен вход в графическую оболочку системы.

Стандартные потоки ввода-вывода этого shell указывают на /dev/pts/X, что мы и видели ранее. Таким образом, прямой связи приложения терминала с std{in,out,err} shell нет.

Можно увидеть, что каждая вкладка в терминале запускает собственную сессию /dev/pts – /dev/pts/0, /dev/pts/1 и т.д.

Мы отправляем команды в shell и видим результат их работы благодаря асинхронному процессу обмена данных между PTY-master и PTY-slave, которым занимается операционная система.

з 40

Следует понимать, что описанная концепция действительна только для современного Linux. Несмотря на внешнюю схожесть, вы не найдете /dev/ptmx в MacOS X, а /dev/ttyX в Android.

Мы всегда можем узнать устройство, с которым ассоциирована наша сессия, и увидеть, кто обращался к мастеру PTY:

~$ tty

/dev/pts/1

~$ lsof /dev/ptmx

COMMAND PID USER FD TYPE DEVICE SIZE/OFF NODE NAME

gnome-ter 10425 rgersh 19u CHR 5,2 0t0 88 /dev/ptmx

xterm 11767 rgersh 5u CHR 5,2 0t0 88 /dev/ptmx

На этот раз lsof мы применили не к запущенному процессу (как делали для дескрипторов shell), а к устройству. Благодаря концепции дескрипторов, есть возможность использовать одно и то же устройство с разными приложениями:

драйвер PTY следит за тем, какой процесс сделал вызов к /dev/ptmx, поэтому для каждого процесса содержимое по одному и тому же пути будет разным (как со стандартными потоками).

## Удаленное подключение

Прелесть абстракций стандартных потоков ввода-вывода дает большую гибкость. Так, рассмотренный пример, в котором shell запускается локальной программой эмулятора терминала, является частным.

Эта программа может быть заменена другой. Например, той, что получает ввод и вывод не от локального оконного сервера, а через защищенный канал от удаленного клиента. Именно так работает SSH (secure shell) сервер.

В этой ситуации именно ssh сервер делает вызов к /dev/ptmx на удаленном сервере, а затем ассоциирует с сессией новый /dev/pts/X.

Так как по умолчанию удаленный пользователь не является аутентифицированным, ssh сервер не запускает сразу shell (так как в этом случае любой мог бы получить доступ), а сначала вызовет login.

# POSIX и \*nix

## POSIX-совместимая Unix-подобная ОС Linux

Bell Labs и AT&T Research создали оригинальную ОС UNIX® в 1970-х годах.

ОС UNIX® сделала популярными:

●иерархические файловые системы с директориями глубокой вложенности;

●единообразное представление устройств для пользователя на уровне файловой системы;

●интерпретатор с возможностью использования одних и тех же команд в интерактивном режиме и скриптах;

●регулярные выражения и многое другое.

Изначальная модель распространения UNIX® привела к появлению множества не полностью совместимых между собой ответвлений этой ОС:

●BSD;

●Solaris;

●AIX;

●и других.

Непереносимость кода между этими родственных ОС показала необходимость создания стандарта. Такой стандарт был разработан и получил название: POSIX – Portable (переносимый) Operating System Interface for Unix.

Кстати, утверждением занималась рабочая группа института IEEE, который и по сей день отвечает за более близкие нам стандарты как 802.11a/b/g/n/ac – WiFi.

Страница

из 42

## Развитие \*nix

В настоящее время ни одна ОС, включая последнюю из UNIX® – System V, не имеет общего исходного кода с оригинальным ПО 70-ых годов. Однако следование концепции дизайна, заданной общим предком, позволяет называть многие системы Unix-подобными, а реализация стандарта POSIX приводит к их схожести с точки зрения пользователя и разработчика.

macOS Big Sur является полностью сертифицированной UNIX®-совместимойОС, а не просто Unix-like :)

GNU/Linux (от GNU is not Unix, мощнейший проект по реализации свободной Unix-like ОС с GPL лицензией) на сегодняшний день является стандартом де-факто в серверном сегменте.

## Linux vs. BSD

Различные варианты BSD не утратили своей актуальности. Их широко используют такие гиганты, как Netflix или WhatsApp. Однако подавляющее большинство интернет-сервисов базируются на одном из многообразных дистрибутивов Linux. Мы не будем уделять времени сравнению разных дистрибутивов, а сконцентрируемся на общих для дистрибутивов основах

# Задачи ОС

## Базовые задачи ОС

●Организация процесса загрузки, предоставление runtime для приложений в виде системных библиотек, интерпретируемых языков, виртуальных машин и т.д. после первоначальной инициализации оборудования.

●Интерфейс работы с оборудованием – программы не хотят знать об особенностях доступа к жесткому диску или сетевому контроллеру, они хотят просто записать данные локально или отправить по сети.

●Виртуализация и планирование использования ресурсов – программы не только не хотят знать о физических особенностях доступа к ресурсу, но и думать о том, что за этот ресурс могут одновременно конкурировать другие запущенные приложения.

●Передача данных локально между приложениями.

●Поддержка многопользовательского режима работы - поддержка большого количества терминалов и соответствующее разграничение прав доступа к ресурсам.

## Дополнительные задачи ОС

●Адаптация к широкому спектру оборудования и возможность расширения функциональности.

●Контроль над программами – принудительное завершение сбойных приложений, при наличии выделенных ресурсов – слежение за их расходом.

●Виртуализация устройств – возможность работы не только с физическим оборудованием хоста, но и предоставление интерфейса для создания псевдо-устройств.

## ОС с точки зрения системного администратора

●Разделение на пользовательскую и ядерную части (юзер-спейс и кернел-спейс).

●Такое разделение является реальным - в Linux есть область памяти, доступная ядру (+ модули ядра, драйверы), работающему с ЦП в привилегированном режиме, и пользовательская часть памяти, которая делится между прикладными программами.

●Управление памятью, планирование (процессы, ввод-вывод локальный и сетевой), большинство файловых систем, доступ к оборудованию работает на уровне ядра и недоступно приложениям из юзер-спейса.

# Системные вызовы,библиотечные вызовы,strace, eBPF

## Системные вызовы – метод общения приложений и ОС

Описанное ранее на практике реализовано механизмом системных вызовов – это интерфейс взаимодействия пользовательских приложений и ядра. Напрямую системные вызовы используются редко, а формат обращения к ним сложен для прикладного программирования. Над системными вызовами написаны библиотеки, которые удобнее для прямого вызова программистами.

В Linux подобной стандартной библиотекой является glibc (GNU Lib C). Зависимость от общей библиотеки libc присутствует практически во всех местах операционной системы от простейших приложений:

~$ ldd $(which ls) | grep libc

libc.so.6 => /lib/x86\_64-linux-gnu/libc.so.6 (0x00007f9ab2e5f000)

и до сложных интерпретаторов:

~$ ldd $(which python3) | grep libc

libc.so.6 => /lib/x86\_64-linux-gnu/libc.so.6 (0x00007f3f75d6a000)

glibc – не единственная стандартная библиотека C. Например, в популярном в контейнерах Alpine Linux вместо нее musl libc – легковесная альтернатива для компактных сборок.

Страница

из 42

## Приложение > библиотечный вызов > системный вызов

Пример CPython:

1. функция print языка Python делает библиотечный вызов libc write:

~$ sotruss python3 -c 'print("Hi, Netology!")' 2>&1 | grep write python3 -> libc.so.6 :\*write(0x1, 0xb96880, 0xe)

2. библиотечный вызов libc, в свою очередь, делает системный вызов write:

~$ strace -e trace=write python3 -c 'print("Hi, Netology!")'>/dev/null

write(1, "Hi, Netology!\n", 14) = 14

+++ exited with 0 +++

Скрипт для отслеживания библиотечных вызовов sotruss из комплекта libc:

~$ dpkg -S $(which sotruss)

libc-dev-bin: /usr/bin/sotruss

strace – программа, использующая подсистему ядра ptrace для отслеживания системных вызовов. Основной недостаток strace – замедление работы приложений, в десятки или даже сотни раз, поэтому в production среде применяйте аккуратно.

## strace на примерах, write

В ядре на сегодня более 380 системных вызовов. Практически все сервисы, которые предоставляет приложениям ОС, – системные вызовы:

●установить сетевое соединение (создать сокет и подключить его к удаленному серверу);

●отправить или получить данные сетевого соединения;

●открыть, прочитать или записать файл (который может располагаться на локальной или удаленной файловой системе любого типа);

●создать новый процесс (фактически, запустить приложение);

●выделить/освободить память (malloc/free) и т.д.

Информация по системным вызовам доступна в man 2 после установки пакета manpages-dev. Еще раз посмотрим на write:

write(1, "Hi, Netology!\n", 14) = 14

man 2 write расскажет, что конкретно мы видим:

write() writes up to count bytes from the buffer starting at buf to thefile referred to by the file descriptor fd.On success, the number of bytes written is returned.

stdout – стандартный вывод с файловым дескриптором номер 1.

## strace на примерах, open

Запишем и откроем файл:

~# echo -n 'Hello Netology!' > /tmp/netology

~# cat read\_something.py

#!/usr/bin/env python3with open('/tmp/netology', 'r') as tmp\_file:

print(tmp\_file.read())

Считываем содержимое из записанного ранее файла, а затем выводим его:

openat(AT\_FDCWD, "/tmp/netology", O\_RDONLY|O\_CLOEXEC) = 3

read(3, "Hello Netology!", 16) = 15

close(3) = 0

write(1, "Hello Netology!\n", 16) = 16

openat присваивает 3 номер fd открываемому файлу /tmp/netology, read прочитал 15 байт, close закрыл 3 дескриптор, write написал содержимое в stdout как в прошлом примере.

## strace, полезные опции

●Часто strace необходимо запустить на уже работающие программы, чтобы понять, почему они не ведут себя должным образом. Для этого есть флаг -p:

~# strace -p $(pgrep -f 'nginx: worker')

strace: Process 25294 attached

epoll\_wait(9, [{EPOLLIN, {u32=4108079352, u64=140097351340280}}], 512, -1) =1

accept4(7, {sa\_family=AF\_INET6, sin6\_port=htons(37288),

sin6\_flowinfo=htonl(0), inet\_pton(AF\_INET6, "::1", &sin6\_addr)...

Программы при работе часто порождают потомков, то есть новые процессы. Чтобы подключившись с -p к процессу и включить при этом трейсинг его потомков, добавьте -f.

~# strace sh -c date

vs.

~# strace -f sh -c date

Бывает, что в системных вызовах вы обнаружите важную для отладки информацию, которая отсутствует в логах.

●По умолчанию вывод строк strace ограничен 32 байтам. Используйте -s с требуемой длиной строки, чтобы увидеть полные сообщения, например -s 65000. В таком случае может пригодиться и -o для записи в файл вместо вывода на экран.

●-y – аннотации к файловым дескрипторам.

●Очень часть системным администраторам помогает strace -e open: показывает чтение конфигурационных файлов из разных мест, недоступные ресурсы и т.д.

## Используйте strace!

Упомянутые утилиты помогут вам в разных реальных ситуациях не раз и полезны для обучения. Большинство вещей, которые мы упоминаем в теории, через strace можно увидеть вживую. Ранее мы упоминали $PATH:

~$ echo $PATH

/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:...

Запустите bash под strace и увидите, как оболочка действительно перебирает эти директории при поиске исполняемого файла:

stat("/usr/local/sbin/bash", 0x7fff921ea4e0) = -1 ENOENT (No such file ordirectory)

stat("/usr/local/bin/bash", 0x7fff921ea4e0) = -1 ENOENT (No such file ordirectory)

stat("/usr/sbin/bash", 0x7fff921ea4e0) = -1 ENOENT (No such file or directory)

stat("/usr/bin/bash", {st\_mode=S\_IFREG|0755, st\_size=1183448, ...}) = 0

## eBPF

eBPF - еще один инструмент (не использует ptrace), позволяющий отследить, что происходит в ОС. Фактически, это целый язык программирования, который генерирует микропрограммы для исполнения ядром и может манипулировать или следить за различными подсистемами.

eBPF не ограничен привязкой к системным или библиотечным вызовам, а также не влияет на производительность. Рекомендуем ознакомиться с полезным набором утилит iovisor BCC. Изучите список готовых eBPF программ – многие вам понадобятся.

Например, можно посмотреть все новые процессы, появляющиеся в системе:

~# execsnoop-bpfcc

PCOMM PID PPID RET ARGS

systemctl 174640 24777 0 /usr/bin/systemctl status nginx

systemctl 174641 24777 0 /usr/bin/systemctl status nginx

# Процессы и треды

Почему в примере с strace openкогда мы открывали в Python файл с файловой системы, был виден вызов open с флагом r для чтения, однако не было подобного open для записи в stdout?

## fork/clone, наследование

На прошлой лекции, рассматривая файловые дескрипторы, мы узнали, что они наследуются новым процессом от своего родителя. Иными словами, запущенный из bash Python пишет на тот же стандартный выход, что был назначен для bash и открыт им же.

Как вы могли догадаться, в реальности за этот механизм тоже отвечают системные вызовы. В частности, семейства вызовов fork (вилка) и clone.

Из man strace про ключ -f:

Trace child processes as they are created by currently traced processes as a result of the fork(2), vfork(2) and clone(2) system calls.

man 2 fork:

fork() creates a new process by duplicating the calling process. The new process is referred to as the child process. At the time of fork() both memory spaces have the same content.

При запуске нового процесса первым действием ОС дублирует тот процесс, который инициировал fork.

## wait родительского процесса

После того как родитель вызвал fork, в памяти находится 2 идентичных процесса в одной точке исполнения. Разница между ними – код возврата системного вызова, по которому процесс должен понять, является ли он родителем (получает число – PID для потомка), или новым дочерним процессом (получает 0 из fork).

Родительский процесс после вызова fork/clone должен сделать wait (waitpid) на PID нового процесса, дождавшись его завершения.

В случае, когда по какой-либо причине родительский процесс не смог обработать код возврата от дочернего процесса, такой дочерний **процесс** становится **зомби** и, завершившись, остается висеть строкой в таблице процессов ядра.

**Процессы сироты** тоже существуют – т.е., родительский процесс которых, был завершен до того, как дочерний отработал.

## exec – замещение клонированного процесса

Резонный вопрос – зачем нам в памяти 2 экземпляра одного процесса? Поэтому третий пункт при запуске нового процесса – вызов exec (execve):

~$ strace -f sh -c ls

clone(child\_stack=NULL, flags=CLONE\_CHILD\_CLEARTID|CLONE\_CHILD\_SETTID|SIGCHLD, child\_tidptr=0x7fae4827b850) = 161515

wait4(-1, strace: Process 161515 attached

<unfinished ...>

[pid 161515] execve("/usr/bin/ls", ["ls"], 0x55bb2b539b48 /\* 29 vars \*/) = 0

...

[pid 161515] +++ exited with 0 +++

<... wait4 resumed>[{WIFEXITED(s) && WEXITSTATUS(s) == 0}], 0, NULL) = 161515

При вызове fork физического копирования данных в памяти не происходит – благодаря механизму виртуальной памяти ядро просто назначает одни и те же участки разным процессам, а copy-on-write уже записывает при необходимости только изменяющиеся данные для нового процесса.

## Код возврата

Мы посмотрели как создаются новые процессы, но не менее важно и то, как они завершают свою работу. Штатный способ завершения – системный вызов exit (exit\_group). Получив его, ОС понимает, что нужно освободить все ресурсы, с которыми процесс работал. Возвращаемый exit code – важный признак, по которому можно сделать вывод об успешности завершенного процесса.

Код возврата 0 сигнализирует об успешности. Отличные от 0 значения зависят от приложения, но так или иначе они указывают, что процесс не отработал штатно.

Код возврата собирает с дочерних процессов wait. В bash узнать exit code можно через служебную переменную ? ($?).

Например, ls удалось успешно сделать листинг директорий в /tmp:

~$ ls /tmp > /dev/null; echo $?

0

А вот доступа в домашний каталог /root у обычного пользователя нет:

~$ ls /root; echo $?

ls: cannot open directory '/root': Permission denied

2

Особенно полезен exit code в сценариях для обработки ошибок.

## Состояния процессов

Процессы могут находиться в разных состояниях: одни активно исполняются, какие-то могли стать зомби, так как не был сделан wait, какие-то могут ожидать выполнения долгой задачи дочернего процесса. Некоторые значимые состояния:

D - uninterruptible sleep (непрерываемый сон, обычно во время IO операций);

R - running и runnable (исполняется или ожидает исполнения);

S - interruptible sleep (обычный спящий процесс, который может быть прерван, ожидает какого-то события);

T - остановлен сигналом управления задачами;

Z - упомянутый ранее зомби-процесс.

## Состояния процессов

uptime – команда, показывающая load average за последние 1, 5 и 15 минут.

Load average (LA) – среднее число процессов в состояниях R и D за рассчитываемый промежуток времени. Обратите внимание, что в Linux D-state влияет на LA. Нередко можно встретить утверждение, что LA - это нагрузка на процессор, но это не соответствует действительности, так как медленное IO под диску также будет влиять на LA напрямую.

LA не нормировано по ядрам. То есть, LA 12 для 12-ядерного хоста – показатель, что система нагружена слабо.

root@service-pod-6c48b4f965-v4s46:/# uptime 14:17:13 up 21 days, 4 min, 0 users, load average: 1.45, 0.92, 0.95

## Просмотр таблицы процессов

ps – process state. По умолчанию показывает “собственные” процессы пользователя в том терминале, из которого ps запущен. Имеет множество ключей, их можно сочетать.

Например, ps aux, где:

a – убирает ограничение о собственных процессах,

u – добавляет расширенный набор часто нужных колонок,

x – убирает ограничение о процессах, запущенных из текущего терминала,

w/ww – убирает ограничение по длине вывода.

~# ps aux | head -n3

USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND

root 1 0.1 1.2 170644 12612 ? Ss Jul05 6:51 /sbin/init

root 2 0.0 0.0 0 0 ? S Jul05 0:00 [kthreadd]

## Просмотр таблицы процессов

Посмотреть процессы конкретного пользователя:

~# ps -u www-data u

USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND

www-data 25294 0.0 0.3 57868 3956 ? S Jul05 0:00 nginx: worker

Обратите внимание, в ps многие суффиксы с минусом и без различаются по смыслу. Расширенный вывод с дополнительными колонками:

~# ps -p 25294 -o pid,user,comm,etime,%mem,%cpu

PID USER COMMAND ELAPSED %MEM %CPU

25294 www-data nginx 2-19:29:30 0.3 0.0

## Можно добавить сортировку:

~# ps -o pid,user,comm,etime,%mem,%cpu --sort=-%mem

## Управление процессами, сигналы

Одна из форм межпроцессного взаимодействия – сигналы.

●Процессу можно послать специальный код, который он может обработать. Нажатия Ctrl + Z или Ctrl + C отправляют сигналы TSTP (temporary stop) и INT (interrupt) соответственно.

●Команда kill - другой способ отправить сигнал:

~# kill -HUP $(cat /var/run/nginx.pid)

Нередко HUP командует процессу перечитать свои конфигурационные файлы без перезапуска. Несмотря на свое название, kill может послать любой сигнал, но по умолчанию применяет сигнал TERM (terminate).

●Сигналы могут быть поданы по имени и по номеру – например, знаменитый kill dash nine (-9): принудительное завершение на случай невозможности выхода программы по штатному сигналу -15.

●SIGKILL и SIGSTOP – два сигнала, которые не могут быть перехвачены процессом и обработаны как-то по желанию разработчика (на все остальные сигналы можно повесить подобные обработчики).

## Треды (потоки/нити) исполнения

Существует много сценариев, когда распараллеливание обработки данных является эффективным. Вызов fork является относительно недорогой операцией, но архитектура программы, подразумевающая несколько процессов, может быть не всегда оптимальна. Такая архитектура – не редкость. Один из самых популярных веб-серверов nginx использует модель с несколькими процессами.

Если разработчик не хочет тратить время на межпроцессное взаимодействие для синхронизации работы, желает использовать более простую модель доступа к общим данным в памяти, ОС предоставляет другой механизм - **тредов**, то есть нескольких потоков исполнения внутри одного процесса.

Основное различие fork и clone – в возможности clone создавать треды. С точки зрения планирования процессорного времени для ОС процессы и потоки идентичны, поэтому может оказаться полезно знать, что какой-то единичный процесс может внутри плодить множество тредов и сильно нагружать CPU (один PID, разные thread id):

~# ps aux | grep mongo[db]

mongodb 220309 1.2 7.5 977304 76112 ? Ssl 11:20 0:01 /usr/bin/mongod--unixSocketPrefix=/run/mongodb --config /etc/mongodb.conf

~# ps aux -T | grep mongodb

mongodb 220309 1 220309 0 23 11:20 ? 00:00:00 /usr/bin/mongod--unixSocketPrefix=/run/mongodb --config /etc/mongodb.conf

mongodb 220309 1 220318 0 23 11:20 ? 00:00:00 /usr/bin/mongod--unixSocketPrefix=/run/mongodb --config /etc/mongodb.conf

## Атрибуты процессов

Мы познакомились с немалым числом атрибутов процесса:

●PID, PPID;

●файловые дескрипторы;

●переменные окружения (также наследуются от родителя);

●обработчик сигналов (системный по-умолчанию или собственный);

●область данных в памяти (мы опускаем подробности в данной лекции, но для корректности: stack, heap, data, text);

●один или более тредов.

## Дополнительные атрибуты процессов

Какие еще есть важные атрибуты, которые мы пока не затронули?

●принадлежность пользователю и группе

●параметры запуска:

~# cat /proc/219695/cmdline

nginx: master process /usr/sbin/nginx -g daemon on; master\_process on;

●принадлежность пространству имен (рассмотрим чуть позже)

●capabilities (расширение концепции привилегированных/непривилегированных процессов)

~# getcap $(which ping)

/usr/bin/ping = cap\_net\_raw+ep

## Файловые дескрипторы, удаленные файлы

На прошлой лекции мы познакомились с файловыми дескрипторами. Одно из свойств, полезных системным администраторам - возможность взаимодействия с открытыми процессом файлами с помощью fd, даже если файл был удален с файловой системы:

~$ python3 -c "import time;f=open('/tmp/do\_not\_delete\_me','r');time.sleep(600);" &

[1] 181248

~$ lsof -p 181248 | grep do\_not\_delete\_me

python3 181248 vagrant 3r REG 253,0 14 1572876 /tmp/do\_not\_delete\_me

~$ cat /tmp/do\_not\_delete\_me

valuable\_data

~$ rm /tmp/do\_not\_delete\_me

~$ cat /tmp/do\_not\_delete\_me

cat: /tmp/do\_not\_delete\_me: No such file or directory

~$ lsof -p 181248 | grep do\_not\_delete\_me

python3 181248 vagrant 3r REG 253,0 14 1572876 /tmp/do\_not\_delete\_me (deleted)

~$ cat /proc/181248/fd/3 > /tmp/do\_not\_delete\_me

~$ cat /tmp/do\_not\_delete\_me

valuable\_data

Будьте осторожны. Если вы удаляете что-то, не зная об использовании файла процессом, то, несмотря на кажущееся отсутствие на файловой системе, файл продолжит занимать место на ней. Нередко такое происходит с логами при некорректно настроенном внешнем процессе ротации, сигнал USR1 может помочь в этом случае (отпустить удаленные файлы) и пересоздать актуальные.

1. Установите средство виртуализации [Oracle VirtualBox](https://www.virtualbox.org/).
2. Установите средство автоматизации [Hashicorp Vagrant](https://www.vagrantup.com/).

С помощью базового файла конфигурации запустите Ubuntu 20.04 в VirtualBox посредством Vagrant:

Создайте директорию, в которой будут храниться конфигурационные файлы Vagrant. В ней выполните vagrant init. Замените содержимое Vagrantfile по умолчанию следующим:

* + Vagrant.configure("2") do |config|
  + config.vm.box = "bento/ubuntu-20.04"

end

Выполнение в этой директории vagrant up установит провайдер VirtualBox для Vagrant, скачает необходимый образ и запустит виртуальную машину.

* + vagrant suspend выключит виртуальную машину с сохранением ее состояния (т.е., при следующем vagrant up будут запущены все процессы внутри, которые работали на момент вызова suspend), vagrant halt выключит виртуальную машину штатным образом.

# **ОС лекция 2**

**$ free -m(в мегабайтах) -h(в гигабайтах)**

Покажет сколько памяти на самом деле используется в системе

~# free -m

total used free shared buff/cache

Mem: 981 175 143 0 661

Swap: 979 4 975

buff/cache включают в себя, в частности, страничный кэш блочных устройств, который ускоряет доступ к горячим данным, но может быть в любой момент сброшен

~# echo 3 >/proc/sys/vm/drop\_caches

~# free -m

total used free shared buff/cache

Mem: 981 178 696 0 106

Swap: 979 4 975

Часть данных, которая оказалась не сброшена, – буферы, т.е. временно занятая для операций ядром или приложениями память.

**top**

– просмотреть список активных процессов в системе с сортировкой по потребляемому процессорному времени (по умолчанию), Shift + F - для интерактивного меню, где s – меняет сортировку и d –включает/выключает отображение колонки.

**atop**

– программа для записи исторического top с заданным интервалом.

Возможно, с учетом современных подходов к сбору метрик и eBPF уже не столь актуальна, но может быть полезна. t / Shift + t для перехода по интервалам, atop -r <log> -b <time> для старта с нужного времени.

echo LOGINTERVAL=10 > /etc/default/atop; systemctl restart atop

**htop**

– более современный вариант top. top был написан задолго до эпохи многоядерных систем, htop представляет нагрузку с учетом визуализации отдельных ядер.

**mpstat**

– аналогичная, по сути, тестовая утилита с более подробной статистикой.

~ # mpstat -P ALL 1

Linux 4.19.102+ (api-6f74c84dfb-wlrrz)\_x86\_64\_(12 CPU)

18:53:40 CPU %usr %nice %sys %iowait %idle

18:53:41 all 9.55 0.00 2.09 0.00 88.11

18:53:41 0 1.00 0.00 0.00 0.00 97.00

18:53:41 5 26.73 0.00 6.93 0.00 65.35

18:53:41 9 30.00 0.00 7.00 0.00 63.00

18:53:41 10 33.33 0.00 6.06 0.00 60.61

Страница

из 28

**iotop**

 – аналогичная top утилита, только для дисковой подсистемы.

Далеко не всегда, даже на специализированном сервере (баз данных) очевидно, какой процесс вызывает высокую загрузку дисков, iotop поможет в этом разобраться.

Само по себе знание о загрузке сетевого линка не говорит нам о том, на что действительно расходуется полоса.

**iftop**

поможет в этом.

iftop показывает сетевое взаимодействие без привязки к процессам. Для подобной статистики запустите

**nethogs <interface>**

Нередко на хосте бывает несколько сетевых интерфейсов, и может быть важно оценить загрузку по ним. В такой ситуации поможет

**sar -n DEV 1**

Приведенные выше инструменты хороши для

интерактивной оценки производительности системы, но они не решают проблемы сбора исторических данных.

*Решение:* актуальная связка **Prometheus + node\_exporter**

С коллектором вы познакомитесь в модуле по мониторингу, но на node\_exporter логично взглянуть на этой лекции.

~/node\_exporter-1.0.1.linux-amd64$ curl -s localhost:9100/metrics

2>/dev/null | grep node\_filesystem\_avail\_bytes | grep mapper

node\_filesystem\_avail\_bytes{device="/dev/mapper/vgvagrant-root",fstype="ext4",mountpoint="/"} 6.075752448e+10

~/node\_exporter-1.0.1.linux-amd64$ df -h | grep mapper/dev/mapper/vgvagrant-root 62G 1.6G 57G 3% /

## **ЯДРО и ДИСТРИБУТИВ**

узнать версию ядра

~# uname -r5.4.0-31-generic

посмотреть дистрибутив

~# cat /etc/issue

Ubuntu 20.04 LTS \n \l

# lsb\_release -a

No LSB modules are available.

Description:Ubuntu 20.04 LTS

посмотреть с какой конфигурацией ядро собрано

~# grep -v ^# /boot/config-$(uname -r) | tail -n2

CONFIG\_PUNIT\_ATOM\_DEBUG=m

CONFIG\_UNWINDER\_FRAME\_POINTER=y

В современных ОС ядра распространяются в форме пакетов, как и другие приложения:

~# dpkg -l | grep linux-image-5

ii linux-image-5.4.0-31-generic 5.4.0-31.35 amd64 Signed kernel image generic

Установочные скрипты сами обновляют записи загрузчика, initrd и vmlinuz (бинарный образ ядра).

## **МОДУЛИ ЯДРА**

По формальной классификации ядро Linux – монолитное. Несмотря на это, есть возможность динамически подгружать и выгружать модули.

Посмотреть загруженные:

~$ lsmod | wc -l

87

Классическая команда для загрузки нового модуля – **insmod**

(insert module), однако есть более удобный современный вариант –

**modprobe**, который умеет автоматически

подгружать зависимости и не требует указания пути:

~$ modinfo virtio\_net | grep depends

depends: net\_failover

... sudo insmod /lib/modules/$(uname -r)/kernel/drivers/net/virtio\_net.ko

insmod: ERROR: could not insert module

/lib/modules/5.4../kernel/drivers/net/virtio\_net.ko: Unknown symbol in module

~$ sudo modprobe virtio\_net

Большинство файлов логов Linux находятся в папке /var/log/ вы можете вывести список файлов логов для вашей системы с помощью команды ls:

**$ ls -l /var/log/**

**Настройка ядра – параметры загрузки**

В части параметров настройки ядро Linux аналогично обычным приложениям - оно принимает строку параметров на вход.

~$ cat /proc/cmdline

BOOT\_IMAGE=/boot/vmlinuz-5.4.0-31-generic root=/dev/mapper/vgvagrant-root

ro net.ifnames=0 biosdevname=0 quiet

В данном примере:

●загрузчик передал путь до образа, из которого ядро загружено;

●на каком устройстве находится корневая файловая система;

●монтировать корневую файловую систему в RO режиме во время загрузки;

●параметры для подсистемы udev об именовании сетевых интерфейсов;

●“тихая” загрузка без сообщений.

Настройки загрузчика специфичны для дистрибутива

В Ubuntu изменить их можно через приведенный файл и

запустив **sudo update-grub**

~$ grep GRUB\_CMDLINE\_LINUX /etc/default/grub

GRUB\_CMDLINE\_LINUX\_DEFAULT="quiet"

GRUB\_CMDLINE\_LINUX="net.ifnames=0 biosdevname=0 "

Если доступа в загруженную ОС нет, параметры ядра можно поменять во время работы загрузчика через его сервисное меню (

init=/bin/sh или single для single user mode).

# **Настройки ядра - sysctl**

Параметров ядра существуют больше тысячи - передавать их в командной строке и каждый раз перезагружаться неудобно. Для решения данной проблемы существует механизм sysctl.

~# sysctl -a | grep 'v4.ip\_forward 'net.ipv4.ip\_forward = 0

~# sysctl -w net.ipv4.ip\_forward=1net.ipv4.ip\_forward = 1

~# sysctl net.ipv4.ip\_forwardnet.ipv4.ip\_forward = 1

Внесенные таким образом изменения не сохранятся после перезагрузки. Чтобы сделать их персистентными, присутствует каталог /etc/sysctl.d. В дистрибутивах обычно ряд настроек заданы отличными от стандартных, поэтому ознакомьтесь с имеющимися там параметрами до внесения своих.

~# grep -v '^#' /etc/sysctl.d/\* | grep = | column -t | head -n2/etc/sysctl.d/10-console-messages.conf:kernel.printk = 4 4 1 7/etc/sysctl.d/10-ipv6-privacy.conf:net.ipv6.conf.all.use\_tempaddr = 2

Применить изменения (без указания -p загрузит /etc/sysctl.conf):

~# sysctl -p /etc/sysctl.d/10-ipv6privacy.conf

net.ipv6.conf.all.use\_tempaddr = 1

net.ipv6.conf.default.use\_tempaddr = 2

## **dmesg и syslog**

Два важных источника информации об ОС

~# dmesg -T | tail -n2

[19:27:11 2020] e1000: eth1 NIC Link is Up 1000 Mbps Full Duplex...

[19:27:11 2020] IPv6: ADDRCONF(NETDEV\_CHANGE): eth1: link becomes ready

1.**dmesg** – инструмент доступа к записям ядра. В dmesg (ключ -T – конвертировать время в человеко-читаемое) присутствует информация о загрузке ОС, об изменении состояния аппаратных ресурсов. Проблемы с дисками, отвалившиеся сетевые карты, ошибки при коррекции ошибок ECC – многие обращающие на себя внимание сообщения найдут свое место здесь.

2. **syslog** – штатный логгер, куда попадают сообщения уже от приложений в юзер-спейсе. Мест, куда приложения могут писать логи, – множество: от локальных файлов до сервисов вроде journaled. Однако/var/log/syslog остается местом, куда стоит заглянуть при оценке состояния системы

# **init – PID 1, процессы ядра**

Мы узнали, что новые процессы в Linux создаются клонированием родительского процесса. Из этого правила есть понятное исключение – первый процесс в системе, который ядро создает самостоятельно и назначает ему PID 1, – init.init – собирательное название для контроллера инициализации ОС, а не конкретная технология. Система инициализации отвечает за:

●достижение корректного состояния на разных этапах загрузки;

●запуска служб самой ОС и прикладных программ в нужном порядке;

●монтирование файловых систем;

●обратный процесс при выключении ОС;

●init должен уметь “усыновлять” процессы, ставшие “сиротами”.

init – не единственный процесс, который создан ядром. Процессы в ps в квадратных скобках – “ядерные” процессы, что видно по 0 размеру RSS памяти в юзер-спейсе.

USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND

root 2 0.0 0.0 0 0 ? S Jul06 0:00 [kthreadd]

root 20 0.0 0.0 0 0 ? S Jul06 0:00 \\_ [oom\_reaper]

# **systemd, systemctl**

Систем инициализации было разработано много, но стандарт для большинства популярных дистрибутивов сегодня – systemd. systemd многие критикуют за несоответствие философии Unix, согласно которой у программы должно быть единственное предназначение, которое она должна выполнять хорошо. systemd же, будучи системой инициализации, сегодня заменяет множественные сервисы ОС: логирование, планировщик задач cron, даже локальный кеширующий DNS и синхронизация времени есть в стандартной поставке, что вообще никак не связано с системой инициализации. Базовые команды:

●systemctl list-units --all (посмотреть все юниты под управлением systemd)

●systemctl status nginx.service (посмотреть статус работы сервиса)сразу же виден путь расположения юнит-файла, который отвечает за сервис (/lib/systemd/system/nginx.service в нашем случае)

●systemctl start/stop/restart/reload nginx(.service можно опустить если типа юнита один)

●systemctl enable/disable nginx (автозапуск сервиса включен/выключен

# **journalctl, systemctl**

●systemctl cat nginx.service

●systemctl edit --full nginx.service

●systemctl daemon-reload (перечитать измененные юнит-файлы самого systemd)

●systemctl list-dependencies nginx.service (посмотреть зависимости сервиса nginx)

●journalctl -b -u nginx.service (посмотреть логи сервиса nginx, которые были записаны с момента загрузки, b от boot)

## **Неймспейсы процессов**

Вы слышали про контейнеры – Docker, LXC и пр. Все базовые технологии для контейнеризации предоставляет само ядро Linux, а известные названия – инструменты, которые делают обращение с ними удобнее.Основная концепция контейнеров – пространства имен. Все, о чем мы с вами говорили, будь то командный интерпретатор bash, процессы-потомки, запущенные из него, – все это тоже находилось в пространстве имен по умолчанию.

~# lsns

NS TYPE NPROCS PID USER COMMAND

4026531835 cgroup 112 1 root /sbin/init 4026531836 pid 112 1 root /sbin/init 4026531837 user 112 1 root /sbin/init 4026531838 uts 110 1 root /sbin/init 4026531839 ipc 112 1 root /sbin/init 4026531860 mnt 1 15 root kdevtmpfs 4026531992 net 112 1 root /sbin/init

Вызов lsns покажет, что типов неймспейсов присутствует несколько. Сейчас нас интересует пространство имен PID. В модуле про виртуализацию/контейнеризацию мы рассмотрим подробнее, как запустить полноценный контейнер только лишь средствами ядра, но и сейчас можно попробовать выполнить элементы этого касательно пространства имен PID.

# **Системный вызов unshare**

Таким нехитрым образом можно организовать полностью изолированный от хостового неймспейс, в котором PID 1 будет иметь команда, которую мы запустили.

~# screen

~# unshare -f --pid --mount-proc /bin/bash

~# ps aux

USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND root 1 0.0 0.3 9836 3980 pts/0 S 07:29 0:00 /bin/bash root 8 0.0 0.3 11476 3316 pts/0 R+ 07:29 0:00 ps aux

Из хостового NS при этом наш процесс имеет совершенно обычный PID:

root 237934 0.0 0.0 8080 596 pts/3 S 07:30 0:00 \\_ unshare -f --pid...

root 237935 0.0 0.3 9836 4012 pts/3 S+ 07:30 0:00 \\_ /bin/bash

Имея привилегии, можно “зайти” в этот namespace с хоста:

~# nsenter --target 237935 --pid –mount

/# ps aux

USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND root 1 0.0 0.3 9836 4012 pts/3 S+ 07:30 0:00 /bin/bash root 28 0.0 0.4 9836 4168 pts/0 S 07:38 0:00 -bash root 37 0.0 0.3 11476 3396 pts/0 R+ 07:38 0:00 ps aux

В /proc/cpuinfo и lscpu показывается только архитектура самого процессора

Дополнительные материалы

Сайт и книги Brendan Gregg, в частности:

●USE-метод

●Широкий список утилит для мониторинга

Пример быстрого интерактивного анализа хоста на практике:

●Netflix

# Файловые системы

## Объекты файловой системы

man 2 stat, /stat \{

Что такое файл? Некоторые важные поля объектов файловой системы:

struct stat { dev\_t st\_dev; /\* ID устройства, содержащего файл \*/ ino\_t st\_ino; /\* номер Inode \*/

mode\_t st\_mode; /\* биты доступа + тип файла \*/

nlink\_t st\_nlink; /\* число hard links (жестких ссылок) \*/

uid\_t st\_uid; /\* User ID – принадлежность пользователю \*/

gid\_t st\_gid; /\* Group ID – принадлежность группе \*/

dev\_t st\_rdev;/\* Device ID – тип устройства, если файл является файлом специального назначения \*/

off\_t st\_size; /\* Размер в байтах \*/

struct timespec st\_atim; /\* Время последнего доступа \*/

struct timespec st\_mtim; /\* Время последнего изменения содержимого \*/

struct timespec st\_ctim; /\* Время последнего изменения метаданных \*/

Уже по структуре данных системного вызова stat мы видим, что файл это не обязательно область на диске с данными. Например, файл может представлять устройство.

timespec – не всегда времена доступа честно обновляются из соображений производительности (опции монтирования).

## Команда stat

~$ stat /etc/hosts

File: /etc/hosts

Size: 198Blocks: 8 IO Block: 4096 regular file

Device: fd00h/64768dInode: 131207 Links: 1

Access: (0644/-rw-r--r--) Uid: ( 0/ root) Gid: ( 0/ root)

Access: xxx 19:51:20.402137404 +0000

Modify: yyy 15:29:25.128599631 +0000

Change: zzz 15:29:25.128599631 +0000

●важнейшее поле – Inode (индексный дескриптор)

●тип файла: regular file (обычный файл)

●1 жесткая ссылка на объект (поле Links)

●Uid – принадлежность пользователю, Gid – принадлежность группе

IO Block – размер блока на файловой системе

~# tune2fs -l /dev/mapper/vgvagrant-root | grep 'Block size'

Block size: 4096

8 Blocks – размер физического блока устройства:

~# cat /sys/block/sda/queue/physical\_block\_size

512

Поэтому, даже при малом размере файла в 198 байт на устройстве выделено 8 блоков по 512 байт (так как минимальный блок ФС – 4096 байт).

## inode – индексный дескриптор

Уникальный в рамках файловой системы контейнер метаданных файла (пара inode+device – однозначный идентификатор объекта в системе).

stat читает данные из inode.

inode – первичный идентификатор объекта на файловой системе.

Имя таковым не является: в рамках одной ФС может быть создано более одного файла с одним и тем же inode и разными именами: число таких ссылок и показывает stat в поле Links. У обычного файла будет только одна ссылка. Создадим более одной ссылки:

~# touch test\_file

~# stat --format=%h test\_file

1

~# ln test\_file this\_is\_hardlink\_for\_test\_file

~# stat --format=%h test\_file

2

~# stat --format=%i test\_file; stat --format=%i

this\_is\_hardlink\_for\_test\_file

1179657

1179657

Если мы удалим оригинальный test\_file, объект продолжит существование на ФС, так как на него будет присутствовать ссылка this\_is\_hardlink\_for\_test\_file:

~# rm test\_file

~# stat --format=%h,%i this\_is\_hardlink\_for\_test\_file

1,1179657

Страница

## inode, имена файлов

●Имена файлов и соответствие имен<>inode хранятся в структуре директории, в которой эти файлы находятся.

●Число inode определяет число объектов, которое может быть создано на ФС.

●Есть ФС, где inode предварительно выделяются при создании (семейство ext); есть те, где inode может быть добавлены динамически (xfs). Об этом важно знать, так как может сложиться ситуация, при которой на ФС есть свободное пространство для создания файлов, но уже нет inode (слишком много файлов).

~# df -h | grep root

/dev/mapper/vgvagrant-root 62G 3.5G 55G 6% /

~# df -i | egrep '(Inodes|root)'

Filesystem Inodes IUsed IFree IUse% Mounted on

/dev/mapper/vgvagrant-root 4104192 100446 4003746 3% /

Одна из встречающихся техник программирования в Linux – изменения поведения программы в зависимости от того, под каким именем ее вызвали.

~# ls -li /usr/bin/bzip2

3407895 -rwxr-xr-x 3 root root 39144 Sep 5 2019 /usr/bin/bzip2

~# find /usr/bin -inum 3407895

/usr/bin/bunzip2

/usr/bin/bzip2

/usr/bin/bzcat

Страница

## dentry (directory entry), директории

Мы узнали, что название файла – запись об именовании inode в структуре, которая представляет директорию. Эта структура называется dentry, когда мы просматриваем содержимое директорий ls, происходит системный вызов getdents64:

~# strace -e getdents64 ls

getdents64(3, /\* 4 entries \*/, 32768) = 112

Так как у любой директории помимо своего имени присутствует и короткая ссылка (.), у директорий всегда не менее двух hardlink.

## man 2 stat, /S\_IFMT

case S\_IFREG: printf("regular file\n"); break; case S\_IFDIR: printf("directory\n"); break; case S\_IFLNK: printf("symlink\n"); break; case S\_IFIFO: printf("FIFO/pipe\n"); break; case S\_IFSOCK: printf("socket\n"); break; case S\_IFBLK: printf("block device\n"); break; case S\_IFCHR: printf("character device\n"); break;

В man по системному вызову stat описаны типы файлов, которые встречаются в Linux:

●regular file, стандартный файл;

●directory, директория;

●symlink – родственная жесткой ссылке сущность. Встречаются названия мягкая ссылка и симлинк. В отличие от hardlink, – symlink это полноценный объект на файловой системе, который, в частности, может переходить ее границы и ссылаться на объекты других ФС. Не увеличивает счетчик ссылок Links и не влияет на сохранность адреса ссылки (аналогия с ярлыком Windows) при удалении;

●pipe – мы с вами уже встречались с пайпами, |, это неименованные пайпы.

~$ mkfifo test\_pipe

~$ ls -l test\_pipe

prw-rw-r-- 1 vagrant vagrant 0 Jul 12 19:03 test\_pipe

~$ echo Hello Netology > test\_pipe &

~$ cat test\_pipe

Hello Netology

## Типы файлов, socket

socket – бывает сетевым (для соединения удаленных хостов) и локальным (для соединения в рамках одного хоста). Сетевые рассмотрим в следующих лекциях. Локальные (доменные) сокеты – еще один тип межпроцессного взаимодействия (в дополнение к уже известным нам именованным и неименованным пайпам, сигналам и разделяемой памяти процессов). Если в сетевых для подключения сокета используется сетевой адрес+порт, в доменном сокете данным адресом является путь на файловой системе.

Плюсы перед pipe:

●сокеты могут быть двунаправленными, тогда как pipe – однонаправленные,

●производительнее,

●много клиентов может писать в один pipe, однако сторона-приемник не может узнать, кто записал в pipe, в socket такой проблемы нет.

Пример: общение nginx с php-fpm через сокет:

~# grep '^listen ' /etc/php/7.4/fpm/pool.d/www.conf

listen = /run/php/php7.4-fpm.sock

~# grep fastcgi\_pass /etc/nginx/sites-enabled/default

fastcgi\_pass unix:/run/php/php7.4-fpm.sock;

~# curl -s 'localhost:81' | grep 'PHP Version' | grep 'td class'<tr><td class="e">PHP Version </td><td class="v">7.4.3 </td></tr>

Страница

## Символические ссылки

root@netology1:/usr/sbin# ls -l vgs\* | head -n2

lrwxrwxrwx 1 root root 3 Feb 13 21:21 vgs -> lvm

lrwxrwxrwx 1 root root 3 Feb 13 21:21 vgscan -> lvm

Так ссылки отображаются на ФС. Они могут быть как относительными (в примере выше), так и абсолютными. symlink’и так же создаются командой ln, но с ключем -s:

~$ touch test\_file

~$ ln -s $HOME/test\_file test\_link

~$ ls -l ~/test\_linklrwxrwxrwx 1 vagrant vagrant 23 xxx 18:29 /home/vagrant/test\_link ->/home/vagrant/test\_file

# Права доступа

## Стандартные права доступа

Биты разрешений: К кому применяются разрешения:

●r – read, чтение. ●владелец

●w – write, запись ●группа владельца

●x – execution, исполнение ●все пользователи

Итого, 9 бит стандартных атрибутов:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Владелец | | | Группа | | | Все | | |
| r | w | x | r | w | x | r | w | x |
| 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |

В десятичной записи:

r: 22 = 4 w: 21 = 2 x: 20 = 1

644 / rw- r-- r-- / 4 + 2 – 4 – 4

755 /rwx r-x r-x / 4 + 2 + 1 – 4 + 1– 4 + 1

## Стандартные права доступа, применимость

Для файлов:

●read для просмотра содержимого файлов

●write для изменения содержимого файлов

●execute для запуска файла в качестве программы (процесса)

Для директорий:

●read для просмотра содержимого директории (ls)

●write для создания или удаления объектов из директории

●execute для chdir в директорию

## Изменение владельца

chown – change owner, изменить владельца/группу:

:/tmp# touch file

:/tmp# chown vagrant file

:/tmp# stat file | grep Uid

Access: (0644/-rw-r--r--) Uid: ( 1000/ vagrant) Gid: ( 0/ root)

С одинарным аргументом – меняет пользователя.

С аргументом вида :group – меняет группу.

:/tmp# chown :vagrant file

:/tmp# stat file | grep Uid

Access: (0644/-rw-r--r--) Uid: ( 1000/ vagrant) Gid: ( 1000/ vagrant)

При вызове вида user:, изменит и пользователя и группу (равнозначно user:user):

:/tmp# touch file2

:/tmp# chown vagrant: file2

:/tmp# stat file2 | grep Uid

Access: (0644/-rw-r--r--) Uid: ( 1000/ vagrant) Gid: ( 1000/ vagrant)

Страница

## Изменение прав доступа

chmod – change mode, изменить биты разрешений доступа.

С десятичной записью полных прав: chmod 0755 file,

или с буквенной записью: chmod u=rwx,g=rx,o=rx file.

Можно добавлять (+) или убирать (-) относительно имеющихся:

:/tmp# stat file | grep Uid

Access: (0755/-rwxr-xr-x) Uid: ( 1000/ vagrant) Gid: ( 1000/ vagrant)

:/tmp# chmod a-x file

:/tmp# stat file | grep Uid

Access: (0644/-rw-r--r--) Uid: ( 1000/ vagrant) Gid: ( 1000/ vagrant)

## umask – user mask, пользовательская маска

Откуда берутся права доступа по умолчанию?

Для директорий штатные права ОС – 0777, для файлов – 0666. Тем не менее, применяя mkdir или touch, вы увидите:

~$ mkdir dir; touch file

~$ stat {dir,file} | grep Uid

Access: (0775/drwxrwxr-x) Uid: ( 1000/ vagrant) Gid: ( 1000/vagrant)

Access: (0664/-rw-rw-r--) Uid: ( 1000/ vagrant) Gid: ( 1000/vagrant)

Ответ: из них вычитается umask (0777 - 0002, 0666 - 0002):

~$ umask

0002

Для привилегированного пользоваться значение umask иное – 0022.

## Дополнительные права доступа

Полное поле атрибутов файла – 12 бит. Перед 9 битами пользовательских разрешений есть еще три:

●setuid

●setgid

●sticky

Логика установки такая же, как с rwx:

|  |  |  |
| --- | --- | --- |
| Не-пользовательские атрибуты | | |
| setuid | setgid | sticky |
| 0 | 0 | 1 |

В десятичной записи:

setuid: 22 = 4 setgid: 21 = 2 sticky: 20 = 1

~$ stat /tmp | grep Uid

Access: (1777/drwxrwxrwt) Uid: ( 0/ root) Gid: ( 0/ root)

## Дополнительные права доступа, применимость

sticky бит используется для директорий, в которые доступна запись многим пользователям, но требуется защитить файлы от чужого вмешательства.

~$ touch /tmp/vagrant\_user\_file

~$ sudo -u www-data touch /tmp/www\_data\_user\_file

~$ ls -l /tmp/\*\_user\_file

-rw-rw-r-- 1 vagrant vagrant 0 Jul 12 09:15 /tmp/vagrant\_user\_file

-rw-rw-r-- 1 www-data www-data 0 Jul 12 09:15 /tmp/www\_data\_user\_file

~$ rm /tmp/vagrant\_user\_file

~$ rm /tmp/www\_data\_user\_file

rm: remove write-protected regular empty file '/tmp/www\_data\_user\_file'?y

rm: cannot remove '/tmp/www\_data\_user\_file': Operation not permitted

setuid / setgid имеют одинаковый механизм работы за исключением применимости к пользователю (U) или группе (G).

Обычно этот бит можно встретить на исполняемых файлах. Пользователь, запускающий такой файл, получает процесс, работающий с правами владельца исполняемого файла (а не своими собственными). Классический пример: вызов утилиты смены пароля passwd:

$ stat $(which passwd) | grep Uid

Access: (4755/-rwsr-xr-x) Uid: ( 0/ root) Gid: ( 0/ root)

passwd требуется обновить файл /etc/shadow, в котором хранятся хеши паролей.Этот файл доступен на запись только root:

~$ stat /etc/passwd | grep Uid

Access: (0644/-rw-r--r--) Uid: ( 0/ root) Gid: ( 0/ root)

suid бит на исполняемом файле passwd дает возможность запустить его с “эффективными” правами root даже обычному пользователю.

~$ ps aux | grep passw[d]

root 242054 0.0 0.3 10464 3344 pts/1 S+ 09:50 0:00 passwdvagrant

~$ grep Uid /proc/242054/status

Uid:1000000 # real, effective

~$ id 0uid=0(root) gid=0(root) groups=0(root)

~$ id 1000uid=1000(vagrant) gid=1000(vagrant)

Этот же пример показывает опасность некорректного употребления suid/guid, – если бы в passwd не было дополнительных проверок, любой пользователь мог бы поменять пароль root, чего, конечно же, сделать нельзя.

~$ passwd root

passwd: You may not view or modify password information for root.

из 44

setgid на директориях имеет другой эффект (setuid обычно не используется). С этим установленным битом вновь создаваемые файлы и директории наследуют группу владельца родительского каталога, а не пользователя, инициировавшего операцию.

~$ mkdir test\_dir

~$ sudo chown :tcpdump test\_dir

~$ stat test\_dir/ | grep Gid

Access: (0775/drwxrwxr-x) Uid: ( 1000/ vagrant) Gid: (65534/ nogroup)

Стандартное поведение без setgid:

~$ mkdir test\_dir/no\_gid; stat $\_ | grep Gid

Access: (0775/drwxrwxr-x) Uid: ( 1000/ vagrant) Gid: ( 1000/ vagrant)

Поведение с установленным setgid:

~$ chmod g+s test\_dir/

~$ stat test\_dir/ | grep Uid

Access: (2775/drwxrwsr-x) Uid: ( 1000/ vagrant) Gid: ( 114/ tcpdump)

~$ mkdir test\_dir/with\_gid; stat $\_ | grep Gid

Access: (2775/drwxrwsr-x) Uid: ( 1000/ vagrant) Gid: ( 114/ tcpdump)

## Дополнительные атрибуты, lsattr/chattr

Помимо стандартных 12 бит для записи разрешений в метаданных файла, могут присутствовать дополнительный атрибуты, способные ввести в затруднение тех, кто о них не знает:

~# echo 'test' > test\_file

-bash: test\_file: Operation not permitted

~# stat test\_file | grep Uid

Access: (0644/-rw-r--r--) Uid: ( 0/ root) Gid: ( 0/ root)

Почему root не может записать данные в файл, которым он владеет и имеет права на запись?

~# lsattr test\_file

----i---------e----- test\_file

Установлен флаг i – immutable, неизменяемый. Полный список расширенных атрибутов – man chattr, chattr -i для удаления immutable:

~# chattr -i test\_file

~# echo 'test' > test\_file

~# echo $?

0

# Организация хранения данных в Linux

