VK Cloud

СТАЧКА

Контейнер под капотом или как 4 syscall'а изменили подход к эксплуатации ПО

Олег Сидоренков, Старший Разработчик

tech

Кто я?

- Разработчик VK Cloud
- Команда KMS (мы нанимаем!)
- Люблю разбираться в инструментах, которые часто использую

tech

План

- Пример контейнера
- Имплементация собственного решения
- Дальнейшее развитие контейнера

Рассмотрим контейнер

Запустим контейнер и рассмотрим
его фичи

```
1   host:~$ sudo docker run --rm -it --entrypoint /bin/sh alpine
2   cont:/$ echo hello
3   hello
```

Файловая система

Не можем видеть вне файловой системы контейнера

```
host:~$ ls -l /opt
total 24
drwx--x--x  4 root    root    4096 May 13 10:17 containerd
drwxr-xr-x  3 root    root    4096 May 21 17:32 google
drwxr-xr-x  4 root    root    4096 Apr 16 10:06 Insomnia
drwxr-xr-x  4 root    root    4096 Aug 20 16:13 Lens
drwxr-xr-x  4 openbao openbao 4096 Apr 21 13:42 openbao
drwxr-xr-x 14 root    root    4096 Apr 25 14:02 zoom
```

Файловая система

Не можем видеть вне файловой системы контейнера

```
host:~$ ls -l /opt
total 24
drwx--x--x  4 root    root    4096 May 13 10:17 containerd
drwxr-xr-x  3 root    root    4096 May 21 17:32 google
drwxr-xr-x  4 root    root    4096 Apr 16 10:06 Insomnia
drwxr-xr-x  4 root    root    4096 Aug 20 16:13 Lens
drwxr-xr-x  4 openbao openbao 4096 Apr 21 13:42 openbao
drwxr-xr-x 14 root    root    4096 Apr 25 14:02 zoom
```

```
cont/: ls -l /opt
total 0
```

Сети

Сеть хоста != сеть контейнера

```
host:~$ sudo docker run --rm -p 80:80 nginx
```

```
host:~$ curl localhost
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
```

Сети

Сеть хоста != сеть контейнера

```
host:~$ sudo docker run --rm -p 80:80 nginx
```

```
host:~$ curl localhost
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
```

```
cont:/$ curl localhost
curl: (7) Failed to connect to localhost port 80...
```

Ресурсы

Можно ограничивать потребление ресурсов

CPU time 92.5 означает работу 92.5% ядра

```
#!/bin/sh

str="GNU Not Unix"

while true; do
        echo $str
        str="GNU stands for ${str}"
done
```

```
host:~$ sudo docker run --rm -it --entrypoint /bin/sh alpine

host:~$ ps aux | grep script.sh | grep -v grep
root      335720 92.5 ... /bin/sh ./script.sh
```

```
host:~$ sudo docker run --rm --cpus=0.5 -it --entrypoint /bin/sh alpine

host:~$ ps aux | grep script.sh | grep -v grep
root      333899 50.3 ... /bin/sh ./script.sh
```

Процессы

Из контейнера не видны хостовые процессы

```
host:~$ ps aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.0  0.0 168200 10336 ?        Ss   09:03   0:01 /sbin/init
root           2  0.0  0.0      0     0 ?        S    09:03   0:00 [kthreadd]
root           3  0.0  0.0      0     0 ?        I<   09:03   0:00 [rcu_gp]
root           4  0.0  0.0      0     0 ?        I<   09:03   0:00 [rcu_par_gp]
...
```

Процессы

Из контейнера не видны хостовые процессы

```
host:~$ ps aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.0  0.0 168200 10336 ?        Ss   09:03   0:01 /sbin/init
root           2  0.0  0.0      0     0 ?        S    09:03   0:00 [kthreadd]
root           3  0.0  0.0      0     0 ?        I<   09:03   0:00 [rcu_gp]
root           4  0.0  0.0      0     0 ?        I<   09:03   0:00 [rcu_par_gp]
...
```

```
cont:~$ ps aux
PID   USER     TIME  COMMAND
    1 root      0:00 /bin/sh
    7 root      0:00 ps aux
```

Процессы

А вот хост видит процессы контейнера!

```
cont:/$ watch echo 'i am alive'
```

```
host:~$ ps aux | grep alive | grep -v grep
root      272688 ... watch echo i am alive
```

```
cont:/$ ps aux | grep alive | grep -v grep
    7 root      0:00 watch echo i am alive
```

Так что же такое контейнер?

Это обычный процесс в Linux с ограничениями по:

- Контексту (процессы, FS, сеть)
- Ресурсам (CPU, RAM, Disk I/O)
- Отдельный userspace

Так что же такое контейнер?

Это обычный процесс в Linux с ограничениями по:

- Контексту (процессы, FS, сеть)
- Ресурсам (CPU, RAM, Disk I/O)
- Отдельный userspace

Давайте напишем свой!

Подготовка

Создадим команду container run, которая будет запускать контейнер

Исходники

```go
func checkErr(err error) {
    if err != nil {
        panic(err)
    }
}

func main() {
    argv := os.Args
    fmt.Println(argv)
    switch argv[1] {
    case "run":
        checkErr(func1())
        checkErr(func2())
        ...
    }
}
```

Exec: теория

Системный вызов exec позволяет “подменять” бинарь

```go
func main() {
    cmd := exec.Command("/bin/echo", "i <3 exec")
    cmd.Stdin = os.Stdin
    cmd.Stdout = os.Stdout
    cmd.Stderr = os.Stderr

    err := cmd.Run()
    if err != nil {
        panic(err)
    }
}
```

```
❯ go run containers/code/scripts/exec.go
i <3 exec
❯
❯
❯
```

Exec: имплементация

На основе примера создаём аналог
docker exec/run

```go
func run(argv []string) error {
    cmd := exec.Command(argv[0], argv[1:]...)
    cmd.Stdin = os.Stdin
    cmd.Stdout = os.Stdout
    cmd.Stderr = os.Stderr

    err := cmd.Run()
    if err != nil {
        return err
    }

    return nil
}
```

Exec: имплементация

На основе примера создаём аналог
docker exec/run

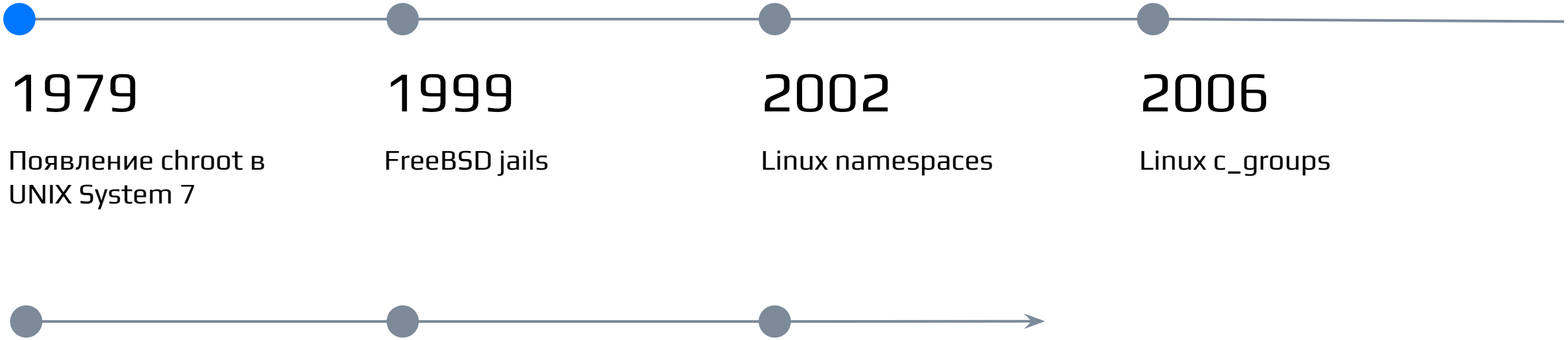
```go
func run(argv []string) error {
    cmd := exec.Command(argv[0], argv[1:]...)
    cmd.Stdin = os.Stdin
    cmd.Stdout = os.Stdout
    cmd.Stderr = os.Stderr

    err := cmd.Run()
    if err != nil {
        return err
    }

    return nil
}
```

```
❯ ./bin/container run /bin/sh
[./bin/container run /bin/sh]
sh-3.2$ echo hi
hi
```

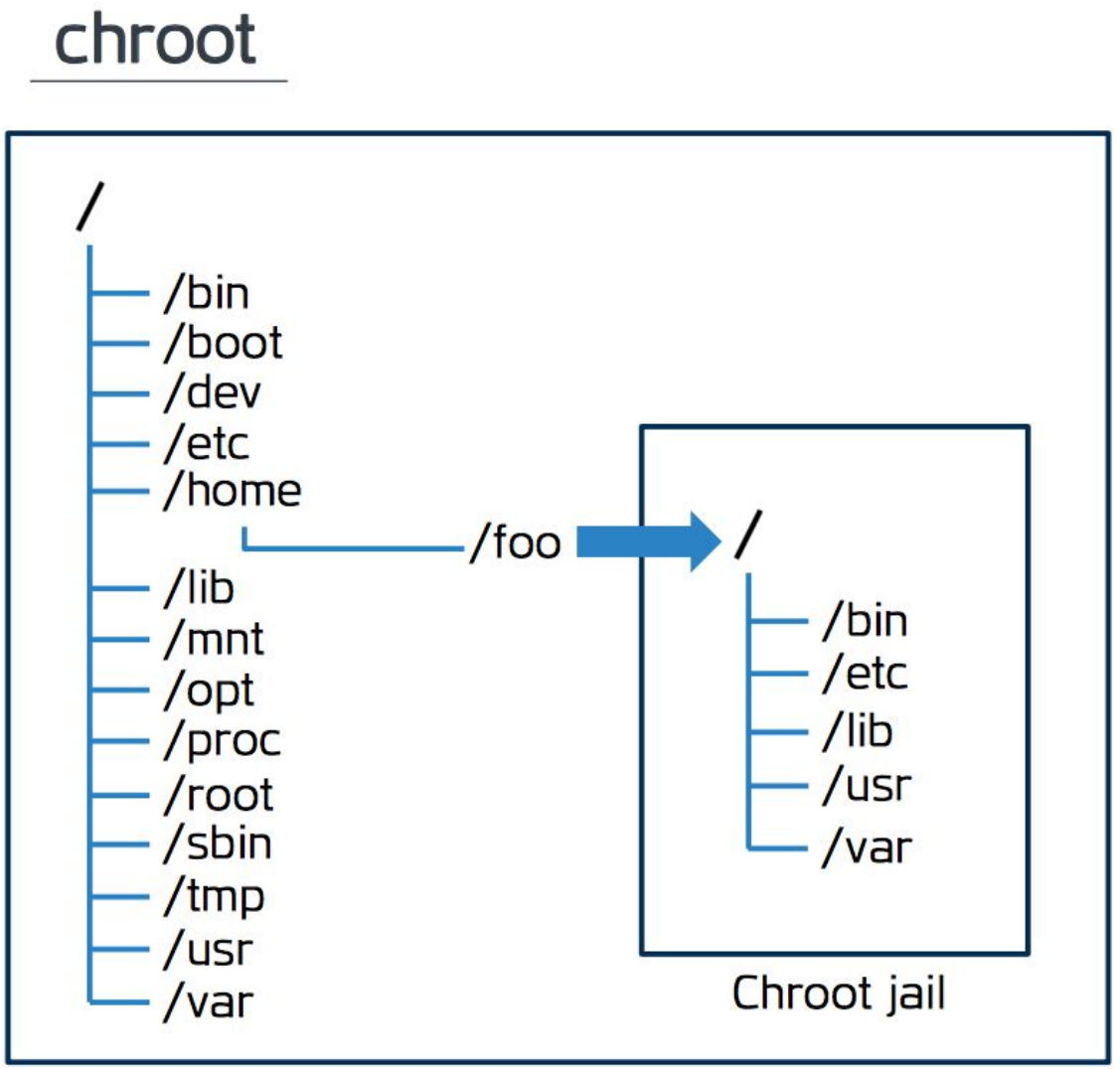
ChRoot: история



ChRoot: теория

Старейший способ изоляции

В хостовой системе создаём каталог, который будет рутом для контейнера



ChRoot: имплементация

Используем Chroot

```go
func changeDirectory(path string) error {
    err := syscall.Chroot(path)
    if err != nil {
        return err
    }
    return nil
}
```

ChRoot: имплементация

Используем Chroot

Бинарник не нашёлся

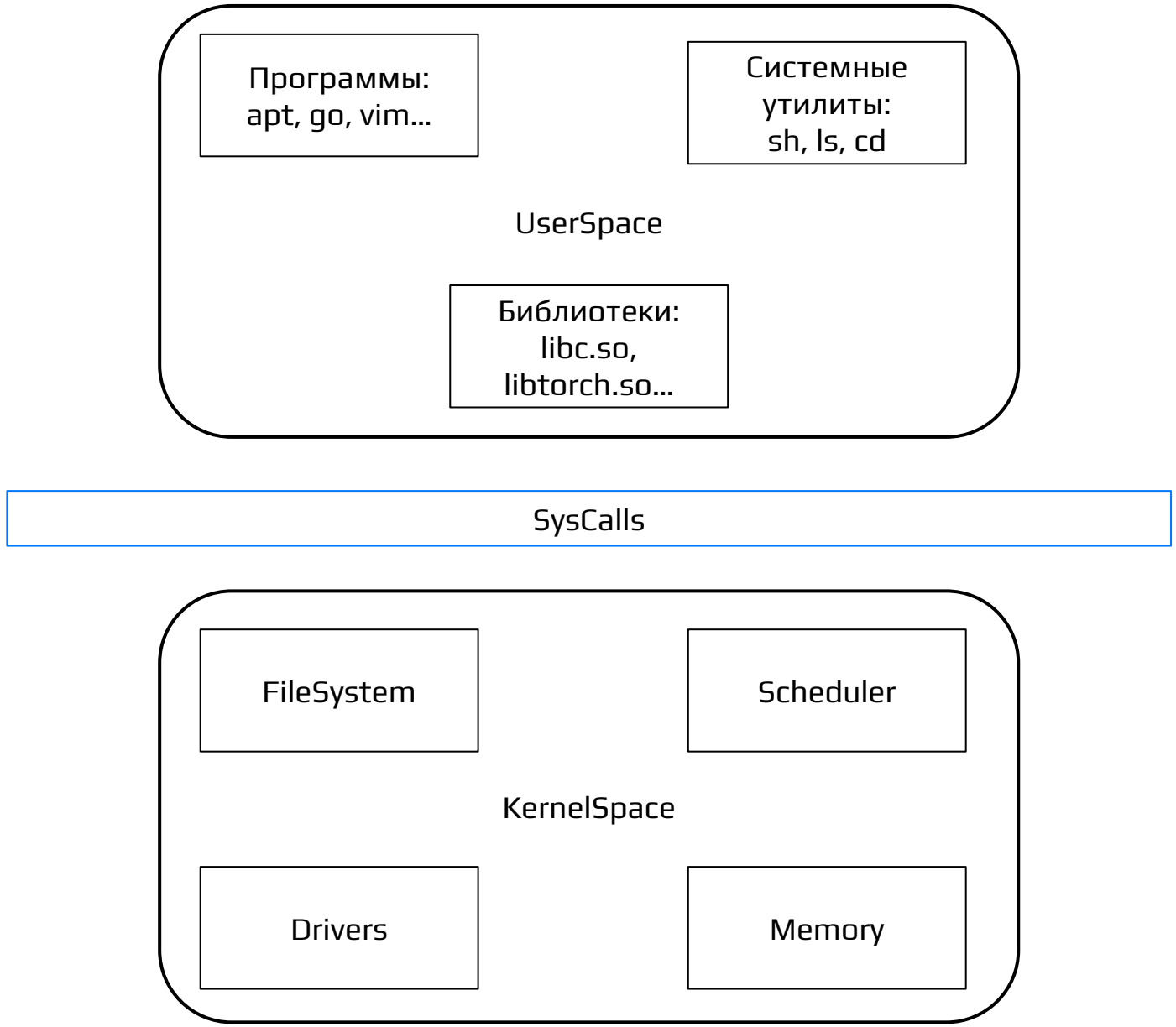
```go
func changeDirectory(path string) error {
    err := syscall.Chroot(path)
    if err != nil {
        return err
    }
    return nil
}
```

```
❯ sudo ./bin/container run /usr/bin/sh
[./bin/container run /usr/bin/sh]
panic: no such file or directory
```

Userspace: теория

KernelSpace - само ядро и относящиеся к нему бинари/библиотеки

UserSpace - набор полезных пользователю утили (бинарей/библиотек)



Userspace: имплементация

Создаём директорию

Заливаем все бинари и библиотеки

Фактически, это и есть docker image

```go
func setupUserspace(path string) error {
    err := syscall.Mkdir(path, 0o777)
    if err != nil {
        fmt.Println(err)
    }

    for _, deps := range []string{"/usr", "/lib", "/lib64"} {
        cmd := exec.Command("cp", "-r", deps, path)
        err = cmd.Run()
        if err != nil {
            return err
        }
    }

    return nil
}
```

```
❯ ls -l /tmp/container
total 0
lrwxrwxrwx  1 root root   7 Oct  1 15:50 lib -> usr/lib
lrwxrwxrwx  1 root root   7 Oct  1 15:50 lib64 -> usr/lib
drwxr-xr-x 10 root root 240 Oct  1 15:50 usr
```

Userspace: имплементация

Можно копировать и выборочно

Заводим бинарники с зависимостями

```
❯ ldd /usr/bin/ls
        linux-vdso.so.1 (0x00007ffd0f76f000)
        libselinux.so.1 => /lib/x86_64-linux-gnu/libselinux.so.1 (0x00007f9baf78c000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f9baf5ab000)
        libpcre2-8.so.0 => /lib/x86_64-linux-gnu/libpcre2-8.so.0 (0x00007f9baf511000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f9baf7fa000)
```

ChRoot: имплементация, вторая попытка

Перемещаемся в каталог

Сетапим PATH

```go
func changeDirectory(path string) error {
    err := syscall.Chdir(path)
    if err != nil {
        return err
    }

    err = syscall.Chroot(path)
    if err != nil {
        return err
    }

    err = os.Setenv("PATH", "/usr/bin")
    if err != nil {
        return err
    }

    return nil
}
```

ChRoot: имплементация, вторая попытка

Перемещаемся в каталог

Сетапим PATH

```go
func changeDirectory(path string) error {
    err := syscall.Chdir(path)
    if err != nil {
        return err
    }

    err = syscall.Chroot(path)
    if err != nil {
        return err
    }

    err = os.Setenv("PATH", "/usr/bin")
    if err != nil {
        return err
    }

    return nil
}
```

```
❯ sudo ./bin/container run /usr/bin/sh
[./bin/container run /usr/bin/sh]
sh-5.2# ls
lib  lib64  usr
sh-5.2# pwd
/
```

Userspace: имплементация

```
❯ sudo ./bin/container run /usr/bin/sh
[./bin/container run /usr/bin/sh]
shell-init: error retrieving current directory: getcwd: cannot access parent directories: No such file or directory
sh-5.2# ls
job-working-directory: error retrieving current directory: getcwd: cannot access parent directories: No such file or directory
bin  container  go.mod  go.sum  Makefile  scripts
sh-5.2# pwd
pwd: error retrieving current directory: getcwd: cannot access parent directories: No such file or directory
pwd: error retrieving current directory: getcwd: cannot access parent directories: No such file or directory
```

Без chdir

Userspace: имплементация

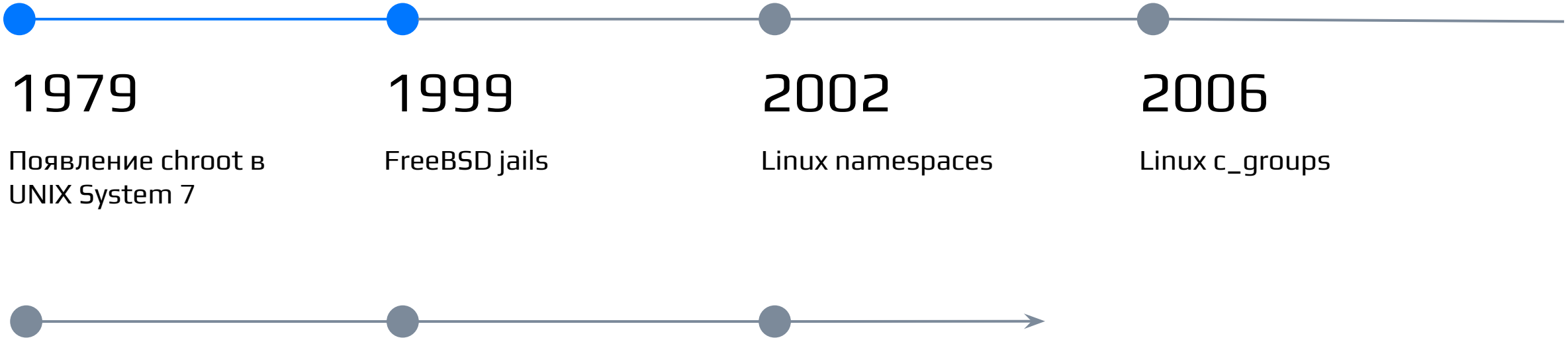
```
❯ sudo ./bin/container run /usr/bin/sh
[./bin/container run /usr/bin/sh]
shell-init: error retrieving current directory: getcwd: cannot access parent directories: No such file or directory
sh-5.2# ls
job-working-directory: error retrieving current directory: getcwd: cannot access parent directories: No such file or directory
bin  container  go.mod  go.sum  Makefile  scripts
sh-5.2# pwd
pwd: error retrieving current directory: getcwd: cannot access parent directories: No such file or directory
pwd: error retrieving current directory: getcwd: cannot access parent directories: No such file or directory
```

Без chdir

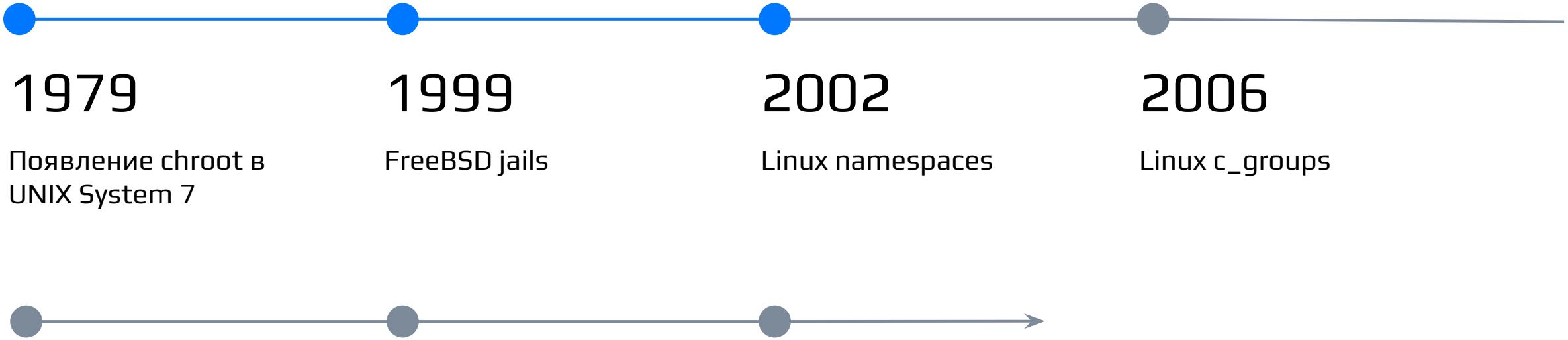
```
❯ sudo ./bin/container run /usr/bin/sh
[./bin/container run /usr/bin/sh]
sh-5.2# ls
lib  lib64  usr
sh-5.2# pwd
/
```

С chdir

Namespaces: история



Namespaces: история



Namespaces: теория

Изолируем:

- Имя хоста и время
- IPC
- Mounts
- Процессы
- Инфу о пользователях
- Net

```
1   struct task_struct {
2       ...
3       struct nsproxy          *nsproxy;
4       ...
5   };
6
7   struct nsproxy {
8       refcount_t count;
9       struct uts_namespace *uts_ns;
10      struct ipc_namespace *ipc_ns;
11      struct mnt_namespace *mnt_ns;
12      struct pid_namespace *pid_ns_for_children;
13      struct net       *net_ns;
14      struct time_namespace *time_ns;
15      struct time_namespace *time_ns_for_children;
16      struct cgroup_namespace *cgroup_ns;
17  };
```

Namespaces: теория

FS - это интерфейс

Не всё в FS является файлом на диске:

- /proc
- /dev

Давайте поменяем данные /proc

```
struct super_operations {
        void (*read_inode) (struct inode *);
        void (*write_inode) (struct inode *, int);
        void (*put_inode) (struct inode *);
        void (*delete_inode) (struct inode *);
        void (*put_super) (struct super_block *);
        void (*write_super) (struct super_block *);
        int (*statfs) (struct super_block *, struct statfs *);
        int (*remount_fs) (struct super_block *, int *, char *);
        void (*clear_inode) (struct inode *);
        void (*umount_begin) (struct super_block *);
};
```

Namespaces: имплементация

Конфижим процесс

Флагов может быть много

```
1  func setNamespace(cmd *exec.Cmd) error {
2      cmd.SysProcAttr = &syscall.SysProcAttr{
3          Cloneflags: syscall.CLONE_NEWPID,
4      }
5      return nil
6  }
```

Namespaces: имплементация

Запустим на хосте процесс

```
❯ watch echo 0 &
[1] 40728
[1]  + 40728 suspended (tty output)  watch echo 0
```

Namespaces: имплементация

Из контейнера он не виден

```
❯ watch echo 0 &
[1] 40728
[1]  + 40728 suspended (tty output)  watch echo 0
```

```
sh-5.2# kill -9 40728
sh: kill: (40728) - No such process
```

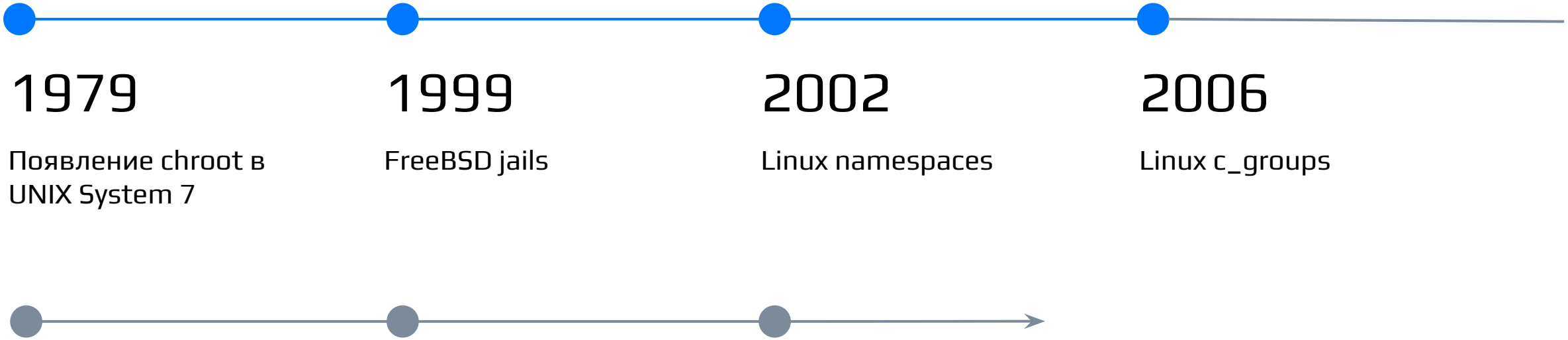
Namespaces: имплементация

Посмотрим, какие процессы у нас есть

Монтируем /proc

```
sh-5.2# mkdir /proc
sh-5.2# mount -t proc /proc proc
sh-5.2# ps
    PID TTY          TIME CMD
      1 ?        00:00:00 sh
      5 ?        00:00:00 ps
sh-5.2# ps aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
0              1  0.0  0.0   7768  4492 ?        S    15:53   0:00 /usr/bin/sh
0              6  0.0  0.0   9704  4304 ?        R+   15:56   0:00 ps aux
sh-5.2#
```

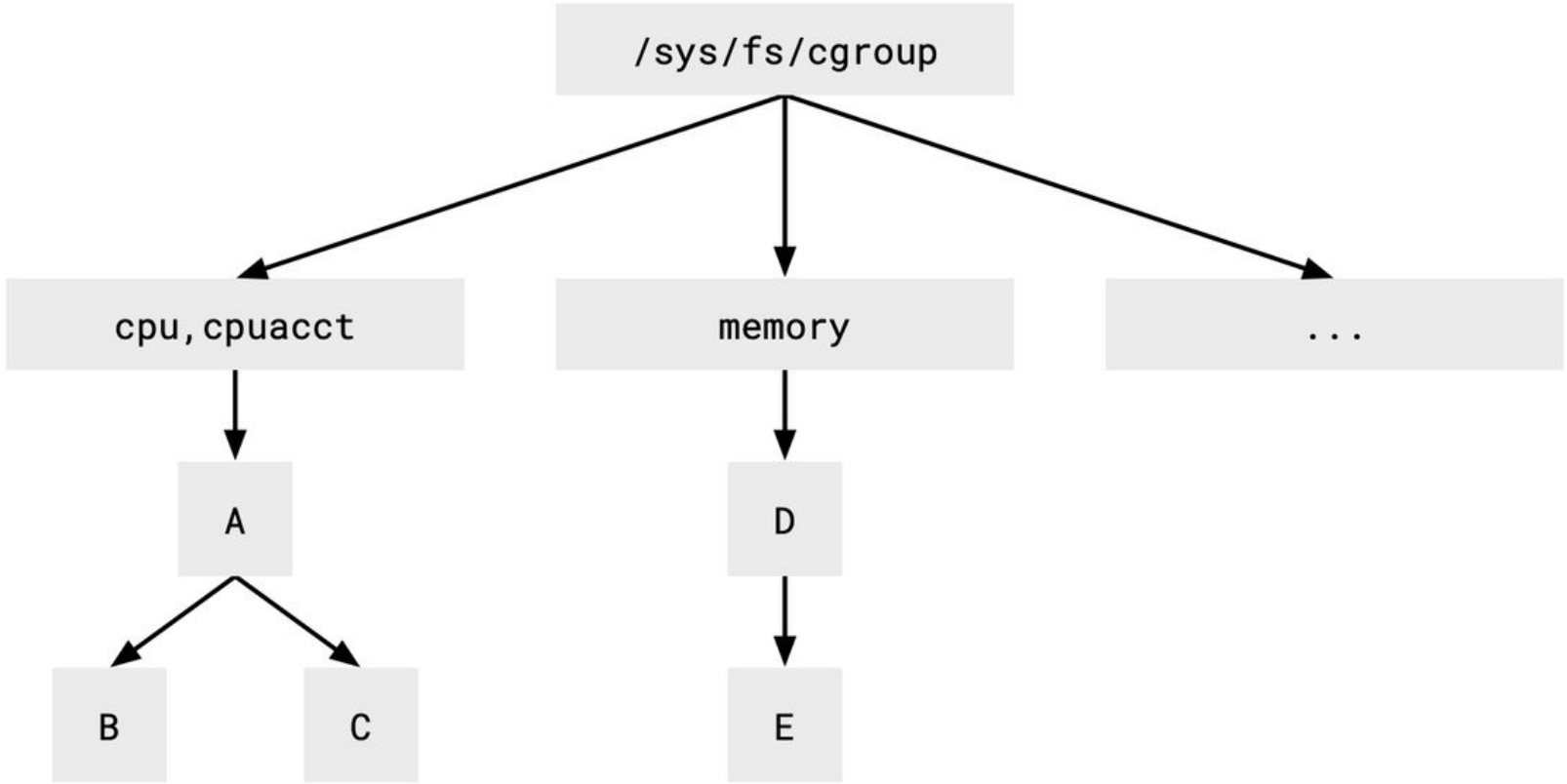
CGroups: история



CGroups: теория

Ограничиваем группу процессов по:

- CPU time
- Memory
- Disk I/O



CGroups: имплементация

Снова запустим бенчмарк

Используется 0.7 CPU

```
❯ ps aux | grep 'benchmark' | grep -v 'grep'
root       46878 69.4  0.0   8292  4024 pts/3    R+   19:08   0:02 /usr/bin/sh ./benchmark.sh
```

CGroups: имплементация

Монтируем /sys/fs/cgroup

Создаём группу container

```go
func setResourceLimits() error {
    err := os.MkdirAll("/sys/fs/cgroup", 0777)
    if err != nil {
        return err
    }

    err = syscall.Mount("/sys/fs/cgroup", "/sys/fs/cgroup", "cgroup2", 0, "")
    if err != nil {
        return err
    }

    path := "/sys/fs/cgroup/container"
    err = os.MkdirAll(path, 0777)
    if err != nil {
        return err
    }
    ...
    return nil
}
```

CGroups: имплементация

Записываем ограничение на 0.5 CPU

Применяется к текущему процессу и его потомкам

```
1  func setResourceLimits() error {
2      ...
3      err = ioutil.WriteFile(
4          filepath.Join(path, "cpu.max"),
5          []byte("50000 100000"),
6          0777,
7      )
8      if err != nil {
9          return err
10     }
11     err = ioutil.WriteFile(
12         filepath.Join(path, "cgroup.procs"),
13         []byte(fmt.Sprintf("%d", os.Getpid())),
14         0777,
15     )
16     if err != nil {
17         return err
18     }
19     return nil
20 }
```

CGroups: имплементация

Записываем ограничение на 0.5 CPU

Применяется к текущему процессу и его потомкам

```
1  func setResourceLimits() error {
2      ...
3      err = ioutil.WriteFile(
4          filepath.Join(path, "cpu.max"),
5          []byte("50000 100000"),
6          0777,
7      )
8      if err != nil {
9          return err
10     }
11     err = ioutil.WriteFile(
12         filepath.Join(path, "cgroup.procs"),
13         []byte(fmt.Sprintf("%d", os.Getpid())),
14         0777,
15     )
16     if err != nil {
17         return err
18     }
19     return nil
20 }
```

CGroups: имплементация

```
sh-5.2# ls /sys/fs/cgroup/container
cgroup.controllers      cgroup.pressure         cpu.idle        cpu.uclamp.max   memory.current       memory.min        memory.stat          memory.zswap.current    pids.max
cgroup.events           cgroup.procs            cpu.max         cpu.uclamp.min   memory.events        memory.numa_stat  memory.swap.current  memory.zswap.max        pids.peak
cgroup.freeze           cgroup.stat             cpu.max.burst   cpu.weight       memory.events.local  memory.oom.group  memory.swap.events   memory.zswap.writeback
cgroup.kill             cgroup.subtree_control  cpu.pressure    cpu.weight.nice  memory.high          memory.peak       memory.swap.high     pids.current
cgroup.max.depth        cgroup.threads          cpu.stat        io.pressure      memory.low           memory.pressure   memory.swap.max      pids.events
cgroup.max.descendants  cgroup.type             cpu.stat.local  irq.pressure     memory.max           memory.reclaim    memory.swap.peak     pids.events.local
```

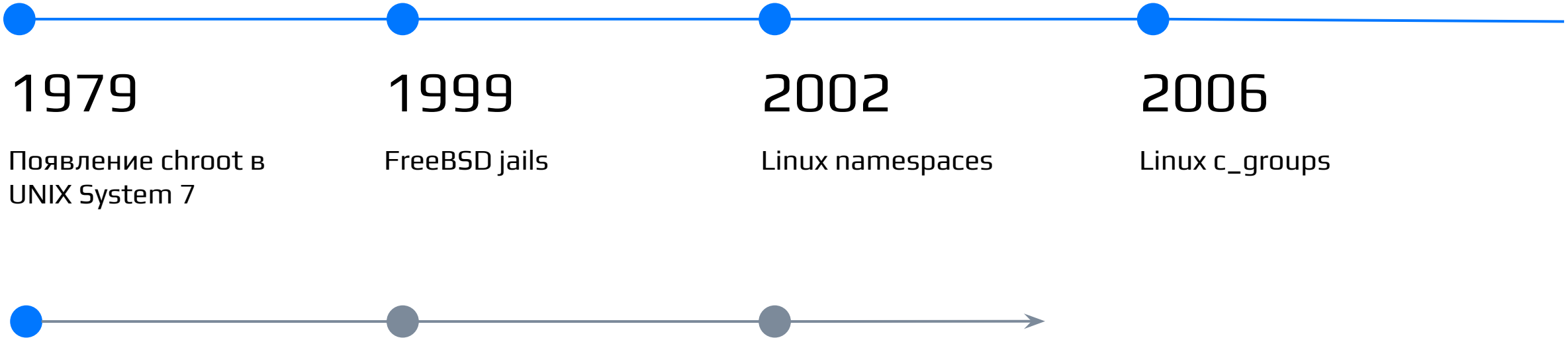
Смонтированные конфиги c_groups

CGroups: имплементация

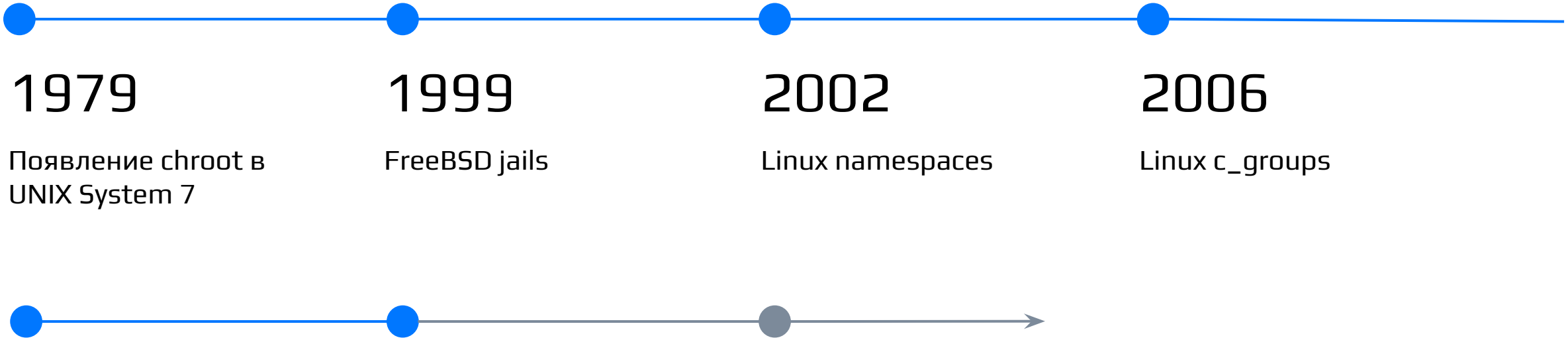
В контейнере скрипт потребляет 0.5 CPU

```
❯ ps aux | grep 'benchmark' | grep -v 'grep'
root       63790 50.1  0.0   8160  3948 pts/3    R+   20:02   0:04 /usr/bin/sh ./benchmark.sh
```

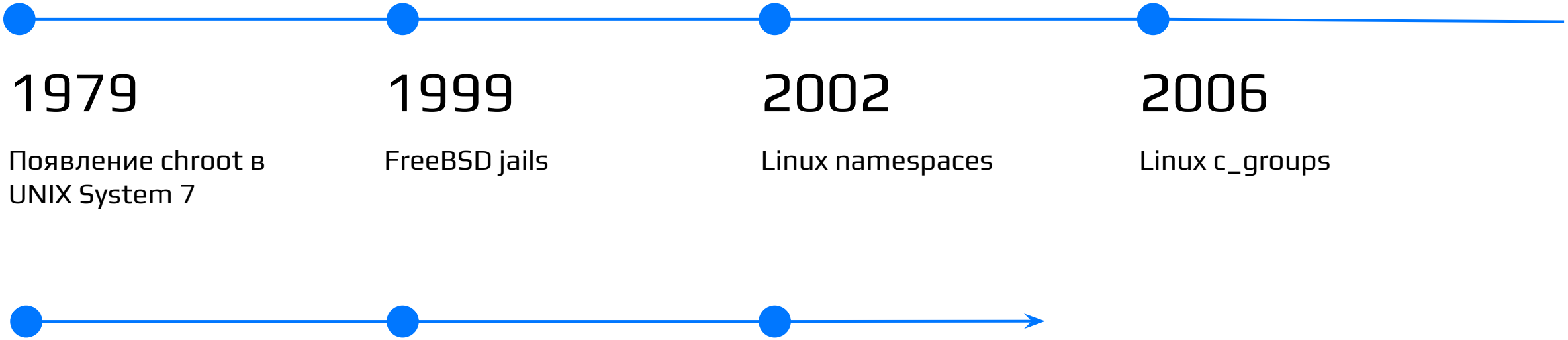
Таймлайн


1979
Появление chroot в UNIX System 7
1999
FreeBSD jails
2002
Linux namespaces
2006
Linux c_groups
2008
LXC
2013
Docker
2014
K8s

Таймлайн


1979
Появление chroot в UNIX System 7
1999
FreeBSD jails
2002
Linux namespaces
2006
Linux c_groups
2008
LXC
2013
Docker
2014
K8s

Таймлайн



1979

Появление chroot в UNIX System 7

1999

FreeBSD jails

2002

Linux namespaces

2006

Linux c_groups

2008

LXC

2013

Docker

2014

K8s

Что дальше?

Что уже сделано:

- Запуск контейнера (exec)
- Изоляция файловой системы (chroot)
- Изоляция других сущностей ОС (namespaces)
- Ограничение ресурсов (c_groups)

Что осталось:

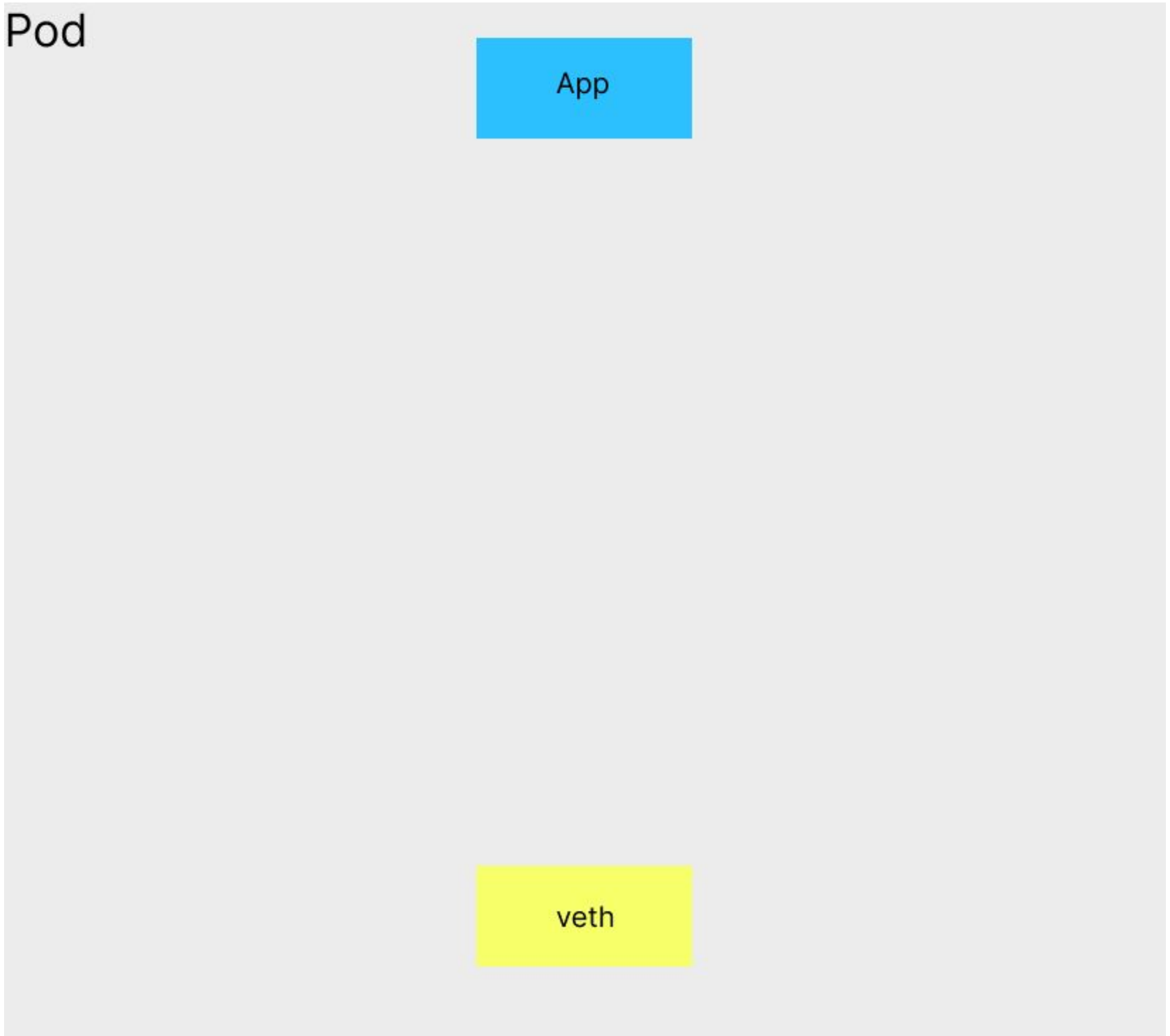
- Сети и изоляция
- Ещё про файловые системы
- VM и CVE (уязвимости)

Networking

Рассмотрим путь запроса

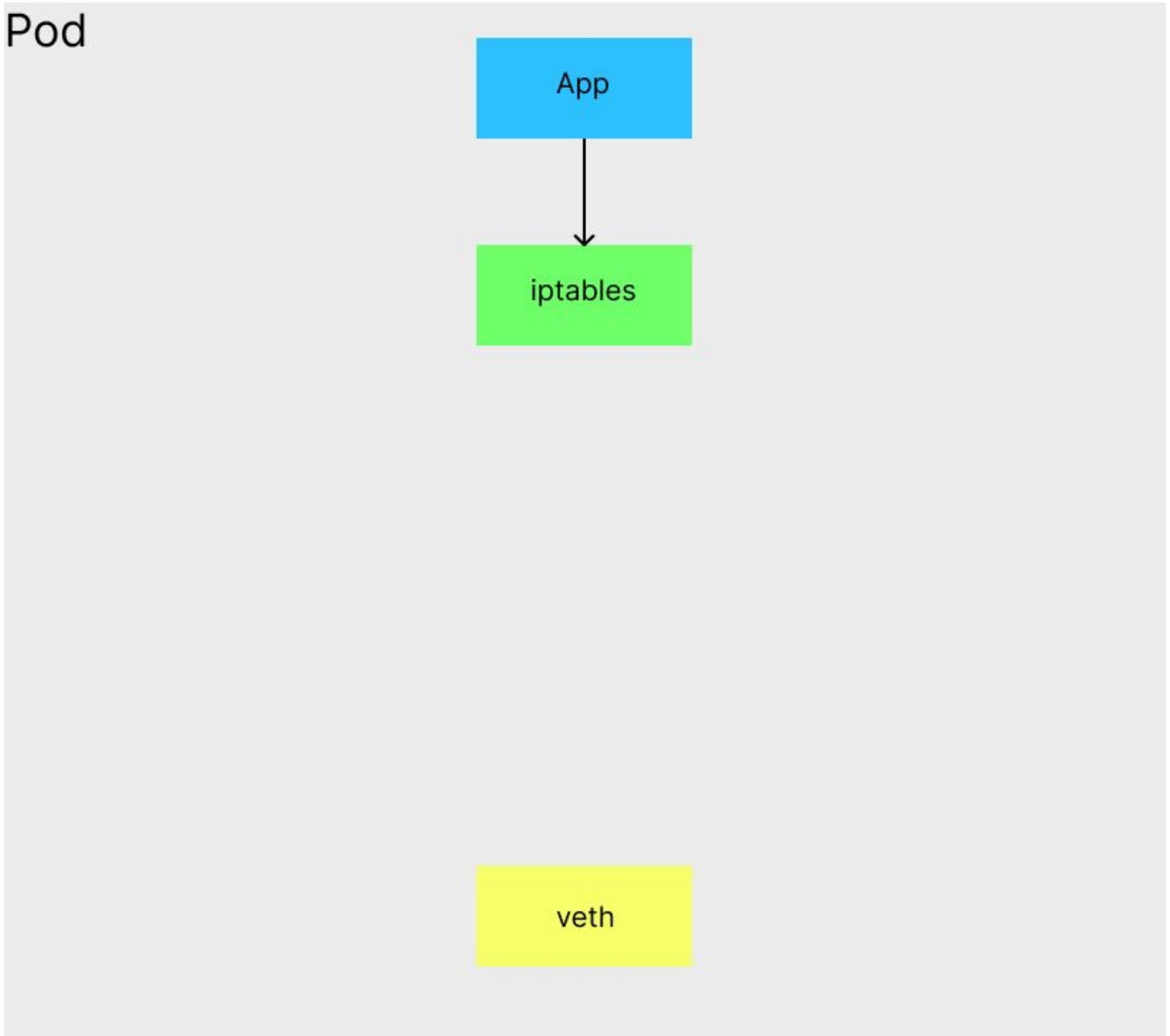
eth - сетевой интерфейс

veth - виртуальный сетевой интерфейс



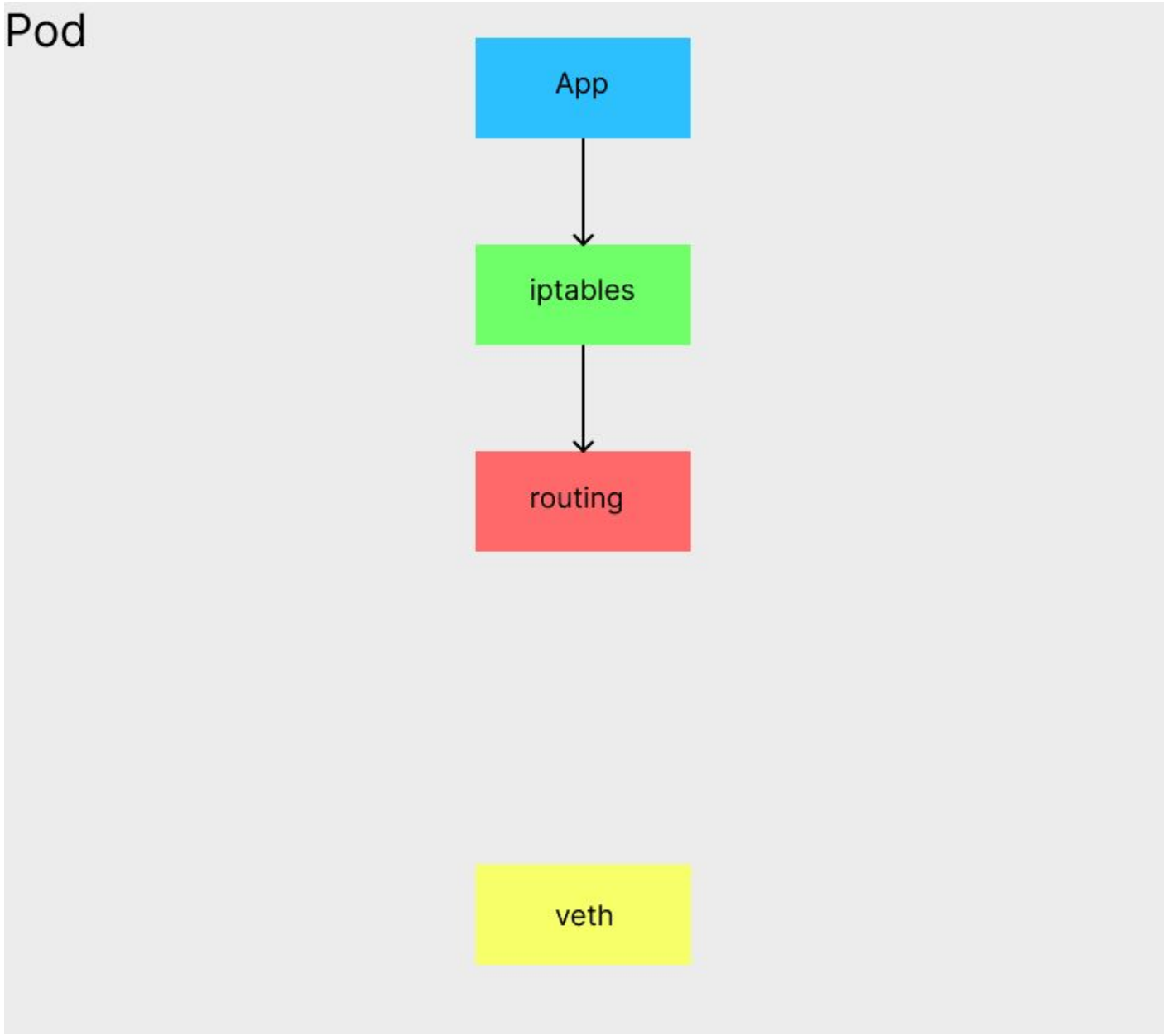
Networking

Входящие правила iptables



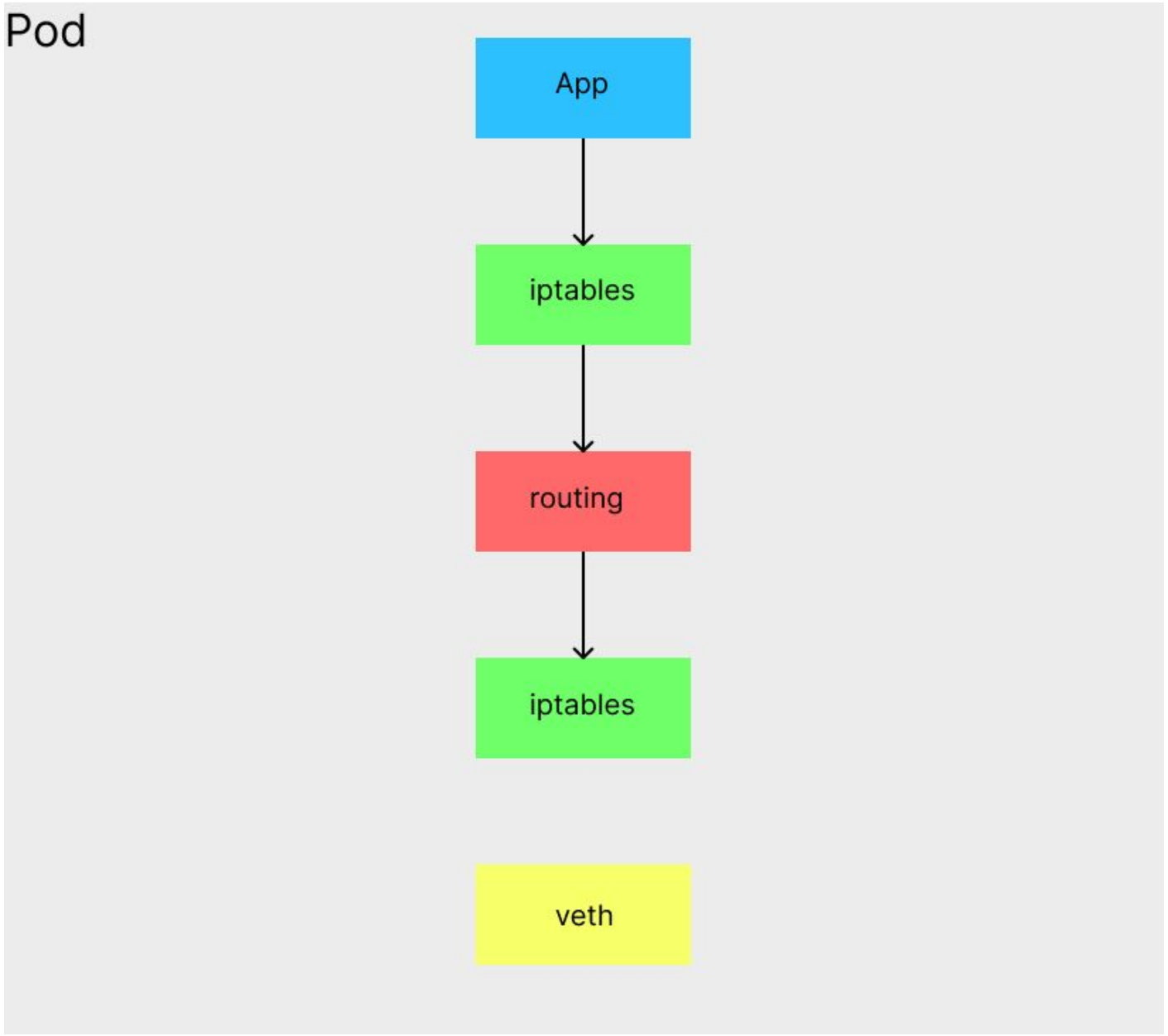
Networking

Маршрутизация ядра



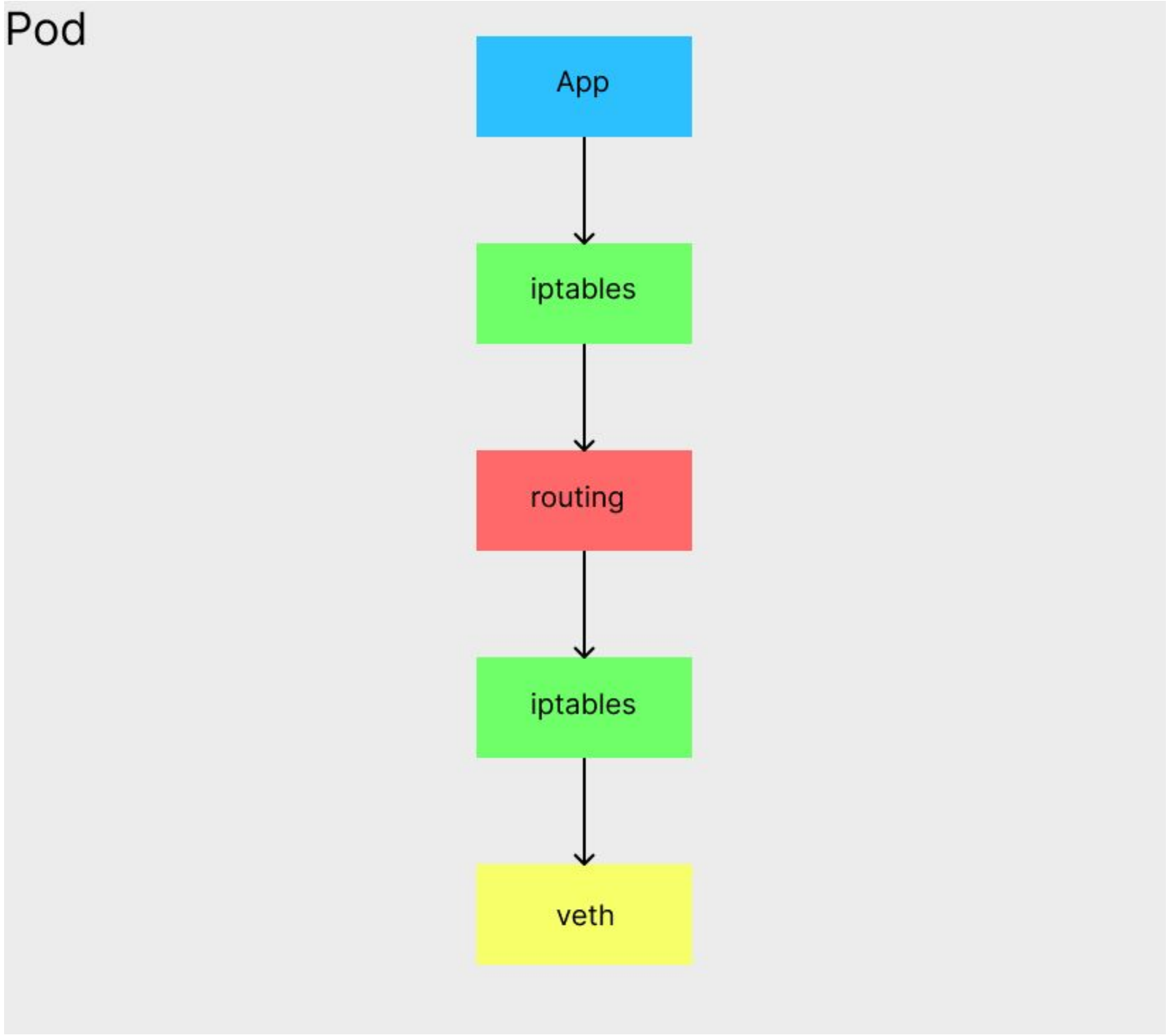
Networking

Исходящие правила iptables



Networking

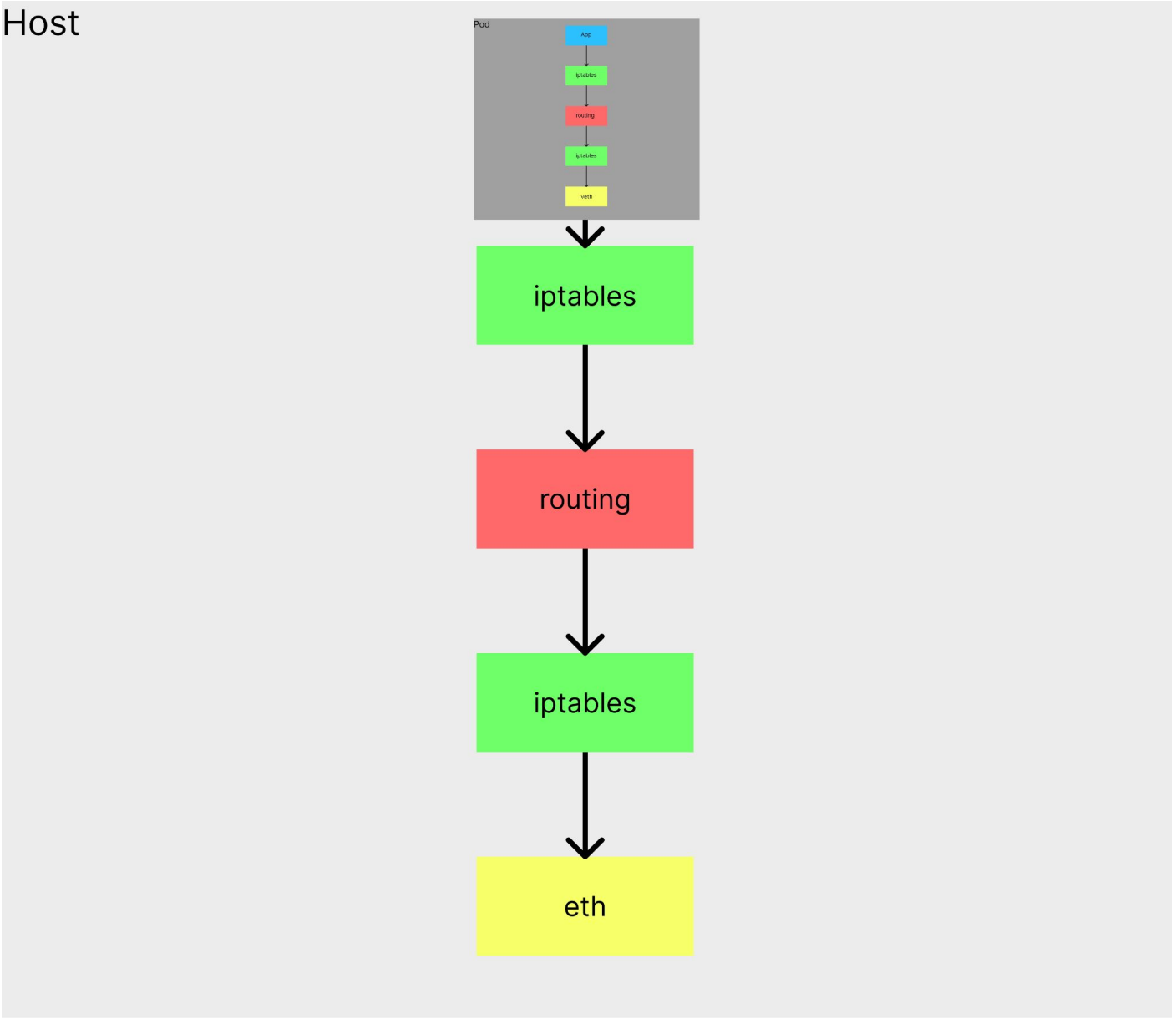
Отправляем на сетевой интерфейс



Networking

На хосте повторяем и подменяем IP

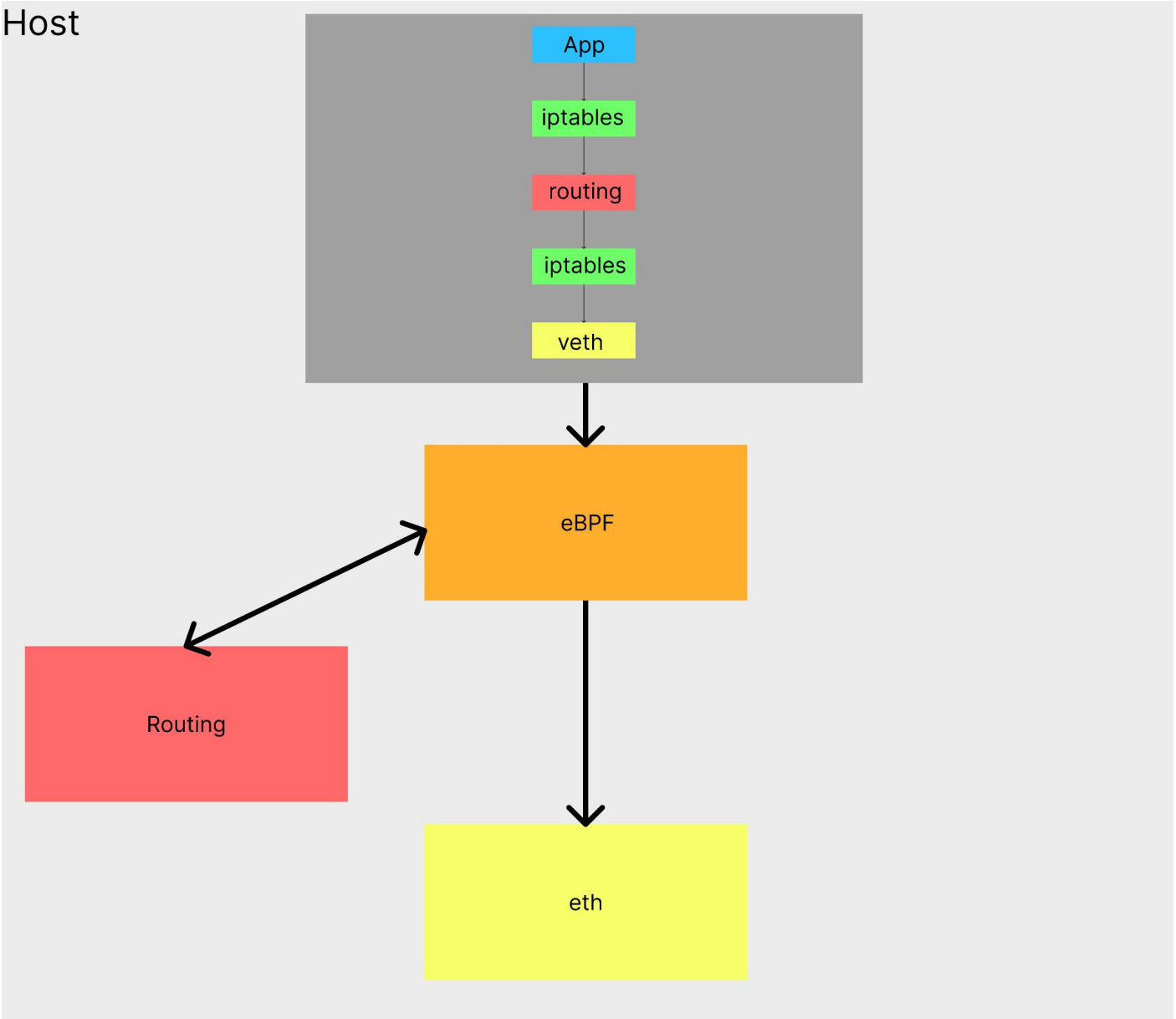
X2 затраты (10-50ms)



tech

Networking

Исправляем с помощью eBPF



tech

Networking

- Подмена IP на выходе

Networking

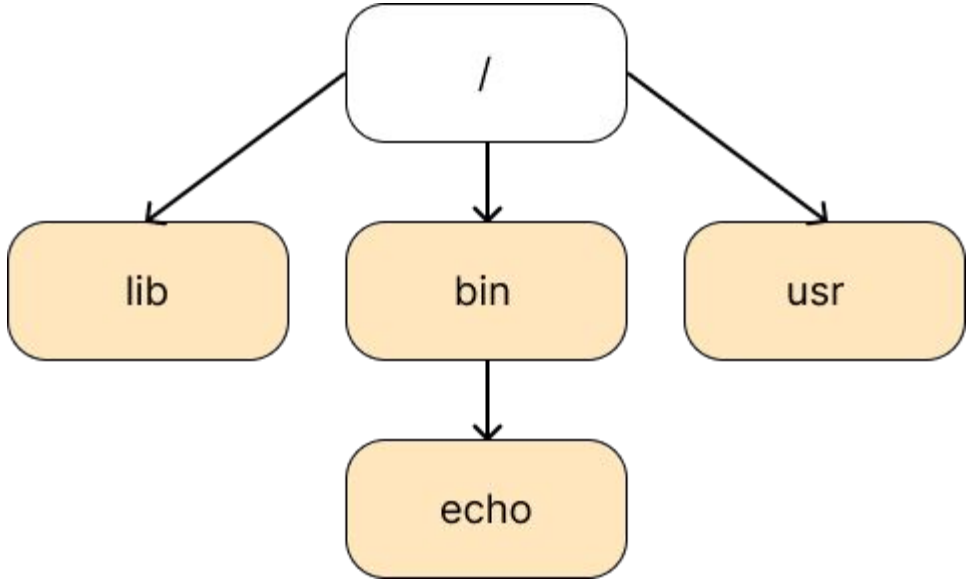
- Подмена IP на выходе
- Изоляция за счёт виртуальных интерфейсов и файервола

Networking

- Подмена IP на выходе
- Изоляция за счёт виртуальных интерфейсов и файервола
- X2 затраты (10-50ms), которые уже можно обойти

File Systems

Проблема: много копирования

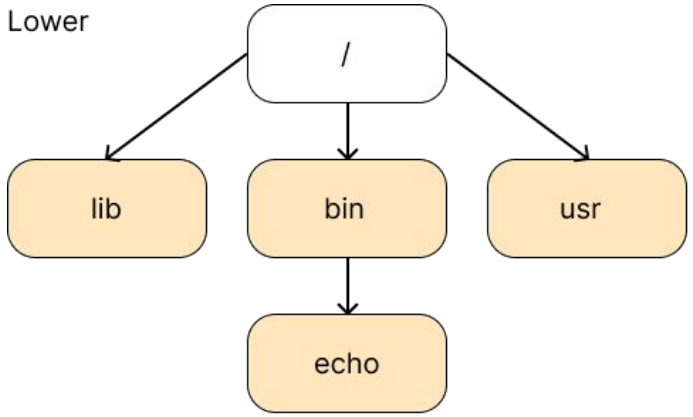


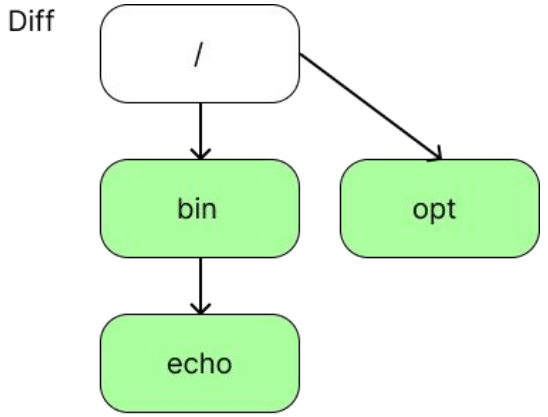
File Systems

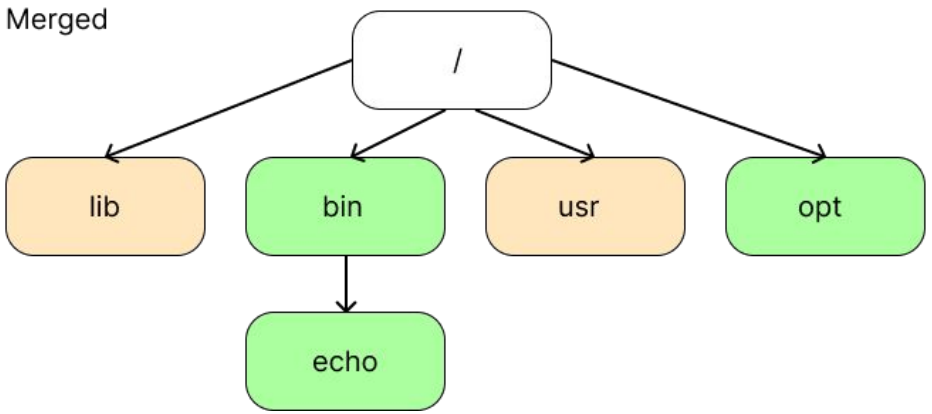
Решение: OverlayFS

- Lower - read-only
- Diff - наши файлы

Итог: Нет лишнего копирования







VM и CVE

	VM	Container
Изоляция	Ядра, Firmware и Hardware раздельны	Общее ядро
Запуск	Минуты	Миллисекунды
IaC	Vagrant + Ansible	K8s
CVE	Умеренно часто. Meltdown	Очень часто. Уязвимости sudo, уязвимости Docker...
Поддерживаемые образы	Любая OS	Linux (есть способы обойти)
Затраты	2-10%	1%

VM и CVE

VM всё ещё нужны:

VM и CVE

VM всё ещё нужны:

- Используются для самих контейнеров (Docker Desktop)

VM и CVE

VM всё ещё нужны:

- Используются для самих контейнеров (Docker Desktop)
- Лучше изоляция и меньше уязвимостей

VM и CVE

VM всё ещё нужны:

- Используются для самих контейнеров (Docker Desktop)
- Лучше изоляция и меньше уязвимостей
- Сценарии, в которых контейнеры не очень подойдут:
  - БД
  - Инстансы на других OS
  - Долгоживущие инстансы со стейтом

VM и CVE

VM всё ещё нужны:

- Используются для самих контейнеров (Docker Desktop)
- Лучше изоляция и меньше уязвимостей
- Сценарии, в которых контейнеры не очень подойдут:
  - БД
  - Инстансы на других OS
  - Долгоживущие инстансы со стейтом

VM и CVE

VM всё ещё нужны:

- Используются для самих контейнеров (Docker Desktop)
- Лучше изоляция и меньше уязвимостей
- Сценарии, в которых контейнеры не очень подойдут:
  - БД
  - Инстансы на других OS
  - Долгоживущие инстансы со стейтом

VM и контейнеры стоит использовать совместно!

Выводы

- Контейнер - отличная абстракция

Выводы

- Контейнер - отличная абстракция
- Контейнер - просто процесс с ограничениями

Выводы

- Контейнер - отличная абстракция
- Контейнер - просто процесс с ограничениями
- Не является виртуальной машиной

Выводы

- Контейнер - отличная абстракция
- Контейнер - просто процесс с ограничениями
- Не является виртуальной машиной
- Настоящие контейнеры - только в Linux

Выводы

- Контейнер - отличная абстракция
- Контейнер - просто процесс с ограничениями
- Не является виртуальной машиной
- Настоящие контейнеры - только в Linux
- Дополнительные траты на контейнеризацию

Что почитать/посмотреть?

Сети в докере:

- https://youtu.be/sEbYhNuDoww

Доклад Лиз Райс с лайвкодингом:

- https://youtu.be/8fi7uSYlOdc

VK Cloud

Олег Сидоренков, Старший Разработчик
t.me/olegdayo

Спасибо за внимание!

Отзыв

tech