



★ 4.84

Оценка

1510.02

Рейтинг

Selectel

IT-инфраструктура для бизнеса



Andrei Yemelianov

15 мар 2016 в 14:57

Механизмы контейнеризации: namespaces

🕒 11 мин 👁 54K

Блог компании Selectel



Последние несколько лет отмечены ростом популярности «контейнерных» решений для ОС Linux. О том, как и для каких целей можно использовать контейнеры, сегодня много говорят и пишут. А вот механизмам, лежащим в основе контейнеризации, уделяется гораздо меньше внимания.

Все инструменты контейнеризации – будь то Docker, LXC или systemd-nspawn, – основываются на двух подсистемах ядра Linux: namespaces и cgroups. Механизм

◆ +36

📖 263



💬 0

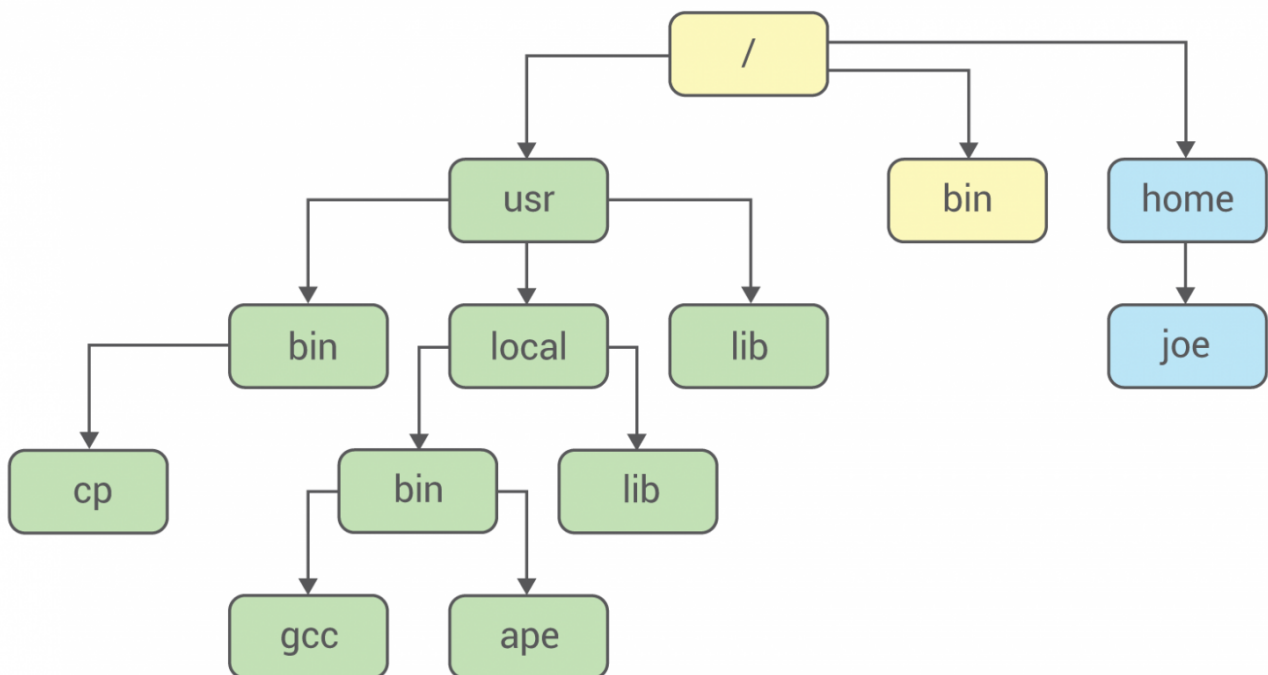
Начнём несколько издалека. Идеи, лежащие в основе механизма пространств имён,

не новы. Ещё в 1979 году в UNIX был добавлен системный вызов `chroot()` – как раз с целью обеспечить изоляцию и предоставить разработчикам отдельную от основной системы площадку для тестирования. Нелишним будет вспомнить, как он работает. Затем мы рассмотрим особенности функционирования механизма пространств имён в современных Linux-системах.

Chroot(): первая попытка изоляции

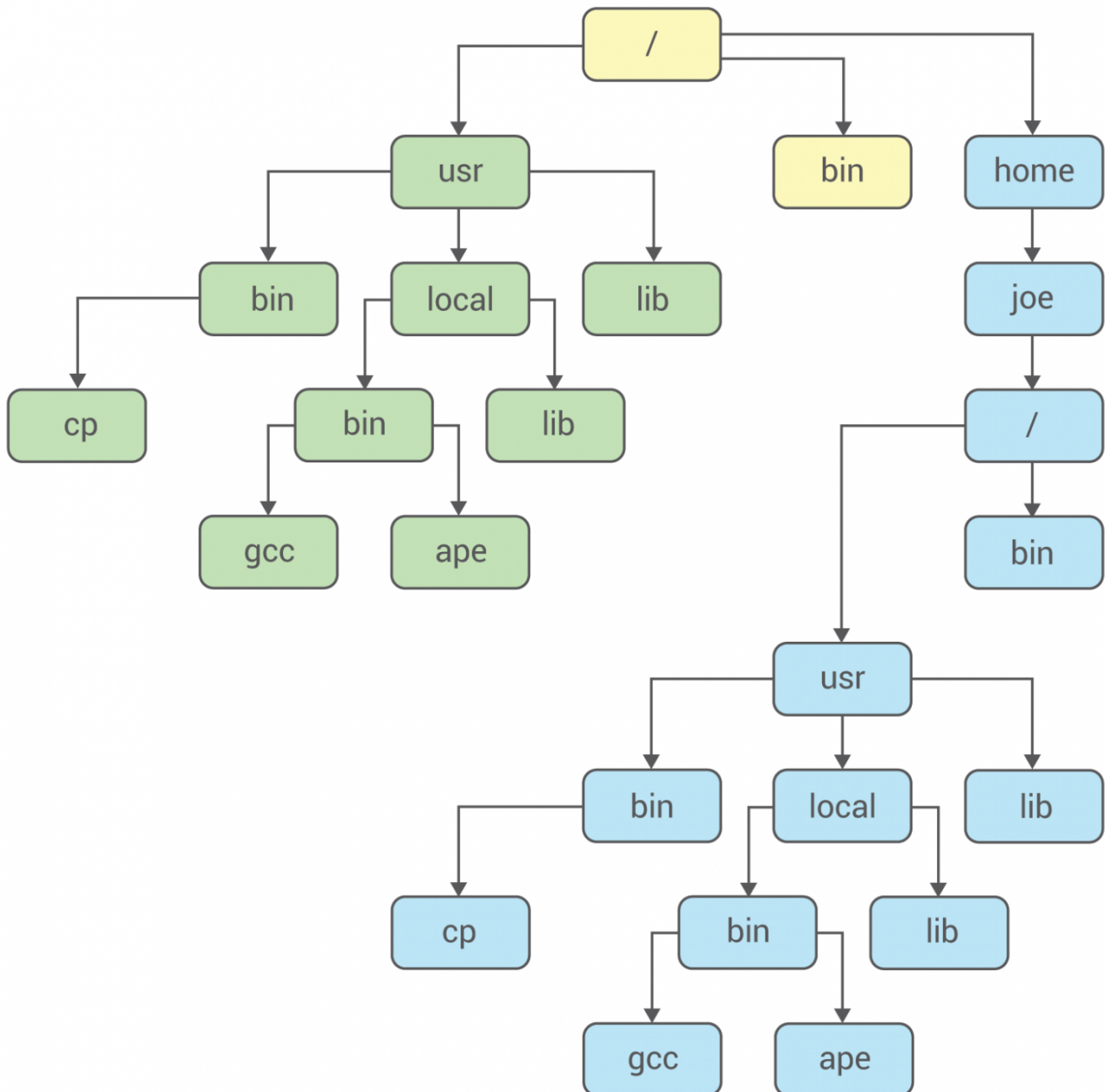
Название `chroot` представляет собой сокращение от `change root`, что дословно переводится как «изменить корень». С помощью системного вызова `chroot()` и соответствующей команды можно изменить корневой каталог. Программе, запущенной с изменённым корневым каталогом, будут доступны только файлы, находящиеся в этом каталоге.

Файловая система UNIX представляет собой древовидную иерархию:



Вершиной этой иерархии является каталог `/`, он же `root`. Все остальные каталоги – `usr`, `local`, `bin` и другие, – связаны с ним.

С помощью `chroot` в систему можно добавить второй корневой каталог, который с точки зрения пользователя ничем не будет отличаться от первого. Файловую систему, в которой присутствует изменённый корневой каталог, можно схематично представить так:



Файловая система разделена на две части, и они никак не влияют друг на друга. Как работает `chroot`? Сначала обратимся к исходному коду. В качестве примера рассмотрим реализацию `chroot` в ОС 4.4 BSD-Lite.

Системный вызов `chroot` описан в файле `vfs_syscall.c`:

```

chroot(p, uap, retval)
    struct proc *p;
    struct chroot_args *uap;
    int *retval;
{
    register struct filedesc *fdp = p->p_fd;
    int error;

```

```

    struct nameidata nd;

    if (error = suser(p->p_ucred, &p->p_acflag))
        return (error);
    NDINIT(&nd, LOOKUP, FOLLOW | LOCKLEAF, UIO_USERSPACE, uap->path, p);
    if (error = change_dir(&nd, p))
        return (error);
    if (fdp->fd_rdir != NULL)
        vrele(fdp->fd_rdir);
    fdp->fd_rdir = nd.ni_vp;
    return (0);
}

```

Самое главное происходит в предпоследней строке приведённого нами фрагмента: текущая директория становится корневой.

В ядре Linux системный вызов `chroot` реализован несколько сложнее (фрагмент кода взят отсюда):

```

SYSCALL_DEFINE1(chroot, const char __user *, filename)
{
    struct path path;
    int error;
    unsigned int lookup_flags = LOOKUP_FOLLOW | LOOKUP_DIRECTORY;
retry:
    error = user_path_at(AT_FDCWD, filename, lookup_flags, &path);
    if (error)
        goto out;

    error = inode_permission(path.dentry->d_inode, MAY_EXEC | MAY_CHDIR);
    if (error)
        goto dput_and_out;

    error = -EPERM;
    if (!ns_capable(current_user_ns(), CAP_SYS_CHROOT))
        goto dput_and_out;
    error = security_path_chroot(&path);
    if (error)
        goto dput_and_out;

    set_fs_root(current->fs, &path);
    error = 0;
dput_and_out:
    path_put(&path);
}

```

```
    if (retry_estale(error, lookup_flags)) {
        lookup_flags |= LOOKUP_REVAL;
        goto retry;
    }
out:
    return error;
}
```

Рассмотрим особенности работы `chroot` в Linux на практических примерах. Выполним следующие команды:

```
$ mkdir test
$ chroot test /bin/bash
```

В результате выполнения второй команды мы получим сообщение об ошибке:

```
chroot: failed to run command '/bin/bash': No such file or directory
```

Ошибка заключается в следующем: не была найдена командная оболочка. Обратим внимание на этот важный момент: с помощью `chroot` мы создаём новую, изолированную файловую систему, которая не имеет никакого доступа к текущей. Попробуем снова:

```
$ mkdir test/bin
$ cp /bin/bash test/bin
$ chroot test
chroot: failed to run command '/bin/bash': No such file or directory
```

Опять ошибка – несмотря на идентичное сообщение, совсем не такая, как в прошлый раз. Простое сообщение было выдан шелл, так как не нашёл нужного исполняемого файла. В примере выше об ошибке сообщил динамический линковщик: он не нашёл необходимых библиотек. Чтобы получить к ним доступ, их тоже нужно копировать в `chroot`. Посмотреть, какие именно динамические библиотеки требуется скопировать, можно так:

```
$ ldd /bin/bash
linux-vdso.so.1 => (0x00007fffd08fa000)
libtinfo.so.5 => /lib/x86_64-linux-gnu/libtinfo.so.5 (0x00007f30289b2000)
libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f30287ae000)
```

```
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f30283e8000)
/lib64/ld-linux-x86-64.so.2 (0x00007f3028be6000)
```

После этого выполним следующие команды:

```
$ mkdir test/lib test/lib64
$ cp /lib/x86_64-linux-gnu/libtinfo.so.5 test/lib/
$ cp /lib/x86_64-linux-gnu/libdl.so.2 test/lib/
$ cp /lib64/ld-linux-x86-64.so.2 test/lib64/
$ cp /lib/x86_64-linux-gnu/libc.so.6 test/lib
$ chroot test
bash-4.3#
```

Теперь получилось! Попробуем выполнить в новой файловой системе, например, команду `ls`:

```
bash-4.3# ls
```

В ответ мы получим сообщение об ошибке:

```
bash: ls: command not found
```

Причина понятна: в новой файловой системе команда `ls` отсутствует. Нужно опять копировать исполняемый файл и динамические библиотеки, как это уже было показано выше. В этом и заключается серьёзный недостаток `chroot`: все необходимые файлы нужно дублировать. Есть у `chroot` и ряд недостатков с точки зрения безопасности.

Попытки усовершенствовать механизм `chroot` и обеспечить более надёжную изоляцию предпринимались неоднократно: так, в частности, появились такие известные технологии, как `FreeBSD Jail` и `Solaris Zones`.

В ядре `Linux` изоляция процессов была усовершенствована благодаря добавлению новых подсистем и новых системных вызовов. Некоторые из них мы разберём ниже.

Механизм пространств имён

Пространство имён (англ. `namespace`) – это механизм ядра `Linux`, обеспечивающий

изоляция процессов друг от друга. Работа по его реализации была начата в версии ядра 2.4.19. На текущий момент в Linux поддерживается шесть типов пространств имён:

| Пространство имён | Что изолирует |
|-------------------|---|
| PID | PID процессов |
| NETWORK | Сетевые устройства, стеки, порты и т.п. |
| USER | ID пользователей и групп |
| MOUNT | Точки монтирования |
| IPC | SystemV IPC, очереди сообщений POSIX |
| UTS | Имя хоста и доменное имя NIS |

Все эти типы используются современными системами контейнеризации (Docker, LXC и другими) при запуске программ.

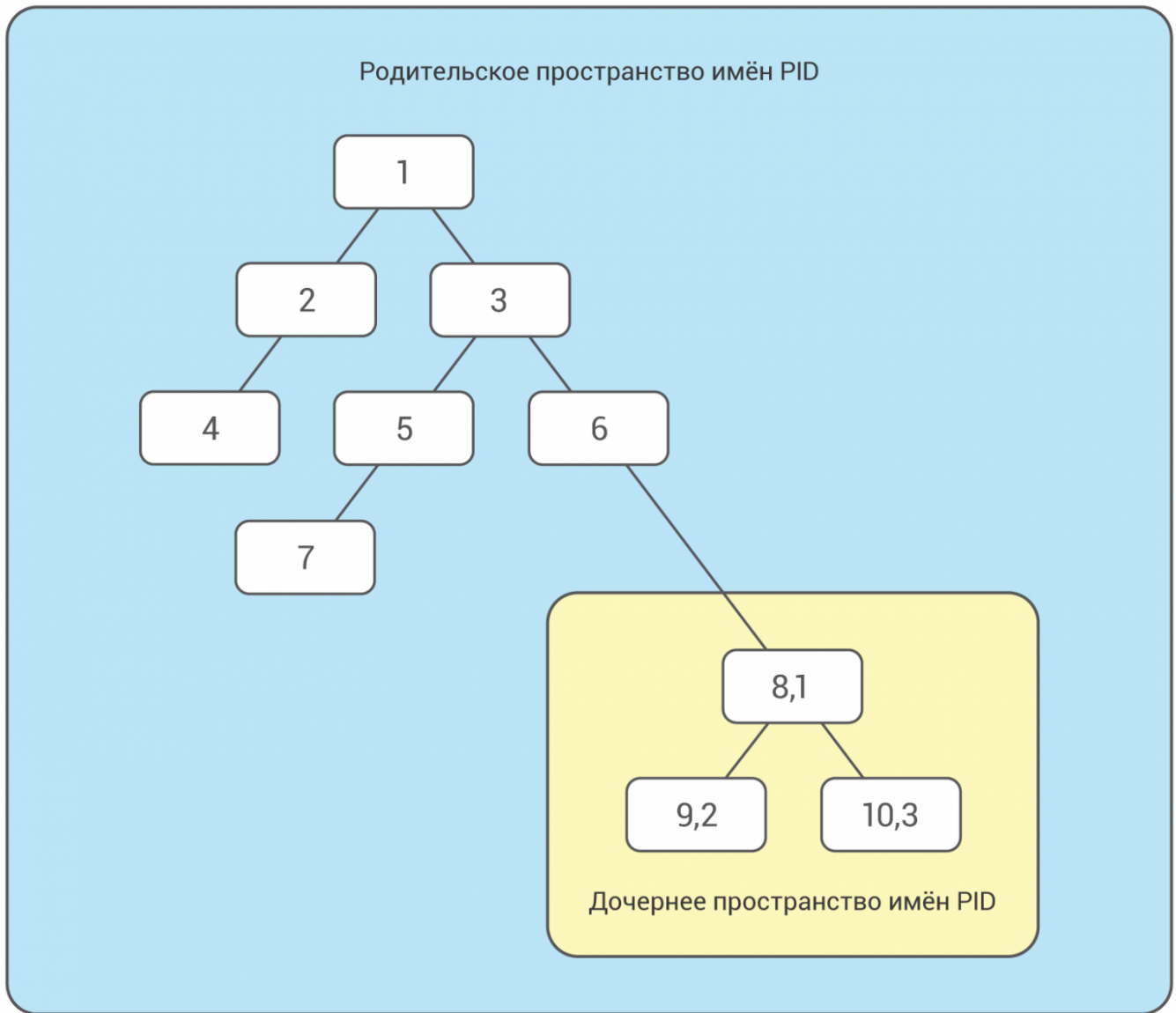
PID: изоляция PID процессов

Исторически в ядре Linux поддерживалось только одно дерево процессов. Дерево процессов представляет собой иерархическую структуру, подобную дереву каталогов файловой системы.

С появлением механизма namespaces стала возможной поддержка нескольких деревьев процессов, полностью изолированных друг от друга.

При загрузке в Linux сначала запускается процесс с идентификационным номером (PID) 1. В дереве процессов он является корневым. Он запускает другие процессы и службы. Механизм namespaces позволяет создавать отдельное ответвление дерева процессов с собственным PID 1. Процесс, который создаёт такое ответвление, являются частью основного дерева, но его дочерний процесс уже будет корневым в новом дереве.

Процессы в новом дереве никак не взаимодействуют с родительским процессом и даже не «видят» его. В то же время процессам в основном дереве доступны все процессы дочернего дерева. Наглядно это показано на следующей схеме:



Можно создавать несколько вложенных пространств имён PID: один процесс запускает дочерний процесс в новом пространстве имён PID, а тот в свою очередь порождает новый процесс в новом пространстве и т.п.

Один и тот же процесс может иметь несколько идентификаторов PID (отдельный идентификатор для отдельного пространства имён).

Для создания новых пространств имён PID используется системный вызов `clone()` с флагом `CLONE_NEWPID`. С помощью этого флага можно запускать новый процесс в новом пространстве имён и в новом дереве. Рассмотрим в качестве примере небольшую программу на языке C (здесь и далее примеры кода взяты отсюда и незначительно нами изменены):

```
#define _GNU_SOURCE
#include <sched.h>
#include <stdio.h>
```



```
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

static char child_stack[1048576];

static int child_fn() {
    printf("PID: %ld\n", (long)getpid());
    return 0;
}

int main() {
    pid_t child_pid = clone(child_fn, child_stack+1048576, CLONE_NEWPID | SIGCHLD, NULL);
    printf("clone() = %ld\n", (long)child_pid);

    waitpid(child_pid, NULL, 0);
    return 0;
}
```

Скомпилируем и запустим эту программу. По завершении её выполнения мы увидим следующий вывод:

```
clone() = 9910
PID: 1
```

Во время выполнения такой маленькой программы в системе произошло много интересного. Функция `clone()` создала новый процесс, клонировав текущий, и начала его выполнение. При этом она отделила новый процесс от основного дерева и создала для него отдельное дерево процессов.

Попробуем теперь изменить код программы и узнать родительский PID с точки зрения изолированного процесса:

```
static int child_fn() {
    printf("Родительский PID: %ld\n", (long)getppid());
    return 0;
}
```

Вывод изменённой программы будет выглядеть так:

```
clone() = 9985  
Родительский PID: 0
```

Строка «Родительский PID: 0» означает, что у рассматриваемого нами процесса родительского процесса нет. Внесём в программу ещё одно изменение и уберём флаг `CLONE_NEWPID` из вызова `clone()`:

```
pid_t child_pid = clone(child_fn, child_stack+1048576, SIGCHLD, NULL);
```

Системный вызов `clone` в этом случае сработал практически так же, как `fork()` и просто создал новый процесс. Между `fork()` и `clone()`, однако, есть существенное отличие, которое следует разобрать детально.

`Fork()` создаёт дочерний процесс, который представляет копию родительского. Родительский процесс копируется вместе со всем контекстом исполнения: выделенной памятью, открытыми файлами и т.п.

В отличие от `fork()` вызов `clone()` не просто создаёт копию, но позволяет разделять элементы контекста выполнения между дочерним и родительским процессами. В приведённом выше примере кода с функцией `clone` используется аргумент `child_stack`, который задаёт положение стека для дочернего процесса. Как только дочерний и родительский процессы могут разделять память, дочерний процесс не может выполняться в том же стеке, что и родительский. Поэтому родительский процесс должен установить пространство памяти для дочернего и передать указатель на него в вызове `clone()`. Ещё один аргумент, используемый с функцией `clone()` – это флаги, которые указывают, что именно нужно разделять между родительским и дочерним процессами. В приведённом нами примере использован флаг `CLONE_NEWPID`, который указывает, что дочерний процесс должен быть создан в новом пространстве имён PID. Примеры использования других флагов будут приведены ниже.

Итак, изоляцию на уровне процессов мы рассмотрели. Но это – всего лишь первый шаг. Запущенный в отдельном пространстве имён процесс все равно будет иметь доступ ко всем системным ресурсам. Если такой процесс будет слушать, например, 80-й порт, это этот порт будет заблокирован для всех остальных процессов. Избежать таких ситуаций помогают другие пространства имён.

NET: изоляция сетей

Благодаря пространству имён NET мы можем выделять для изолированных процессов

собственные сетевые интерфейсы. Даже loopback-интерфейс для каждого пространства имён будет отдельным.

Сетевые пространства имён можно создавать с помощью системного вызова `clone()` с флагом `CLONE_NEWNET`. Также это можно сделать с помощью `iproute2`:

```
$ ip netns add netns1
```

Воспользуемся `strace` и посмотрим, что произошло в системе во время приведённой команды:

```
.....
socket(PF_NETLINK, SOCK_RAW|SOCK_CLOEXEC, 0) = 3
setsockopt(3, SOL_SOCKET, SO_SNDBUF, [32768], 4) = 0
setsockopt(3, SOL_SOCKET, SO_RCVBUF, [1048576], 4) = 0
bind(3, {sa_family=AF_NETLINK, pid=0, groups=00000000}, 12) = 0
getsockname(3, {sa_family=AF_NETLINK, pid=1270, groups=00000000}, [12]) = 0
mkdir("/var/run/netns", 0755) = 0
mount("", "/var/run/netns", "none", MS_REC|MS_SHARED, NULL) = -1 EINVAL (Invalid
mount("/var/run/netns", "/var/run/netns", 0x4394fd, MS_BIND, NULL) = 0
mount("", "/var/run/netns", "none", MS_REC|MS_SHARED, NULL) = 0
open("/var/run/netns/netns1", O_RDONLY|O_CREAT|O_EXCL, 0) = 4
close(4) = 0
unshare(CLONE_NEWNET) = 0
mount("/proc/self/ns/net", "/var/run/netns/netns1", 0x4394fd, MS_BIND, NULL) = 0
exit_group(0) = ?
+++ exited with 0 +++
```

Обратим внимание: здесь для создания нового пространства имён использован системный вызов `unshare()`, а не уже знакомый нам `clone`. `Unshare()` позволяет процессу или треду отделять части контекста исполнения, общие с другими процессами (или тредами).

Как можно помещать процессы в новое сетевое пространство имён?

Во-первых, процесс, создавший новое пространство имён, может порождать другие процессы, и каждый из этих процессов будет наследовать сетевое пространство имён родителя.

Во-вторых, в ядре имеется специальный системный вызов – `setns()`. С его помощью можно поместить вызывающий процесс или тред в нужное пространство имён. Для этого требуется

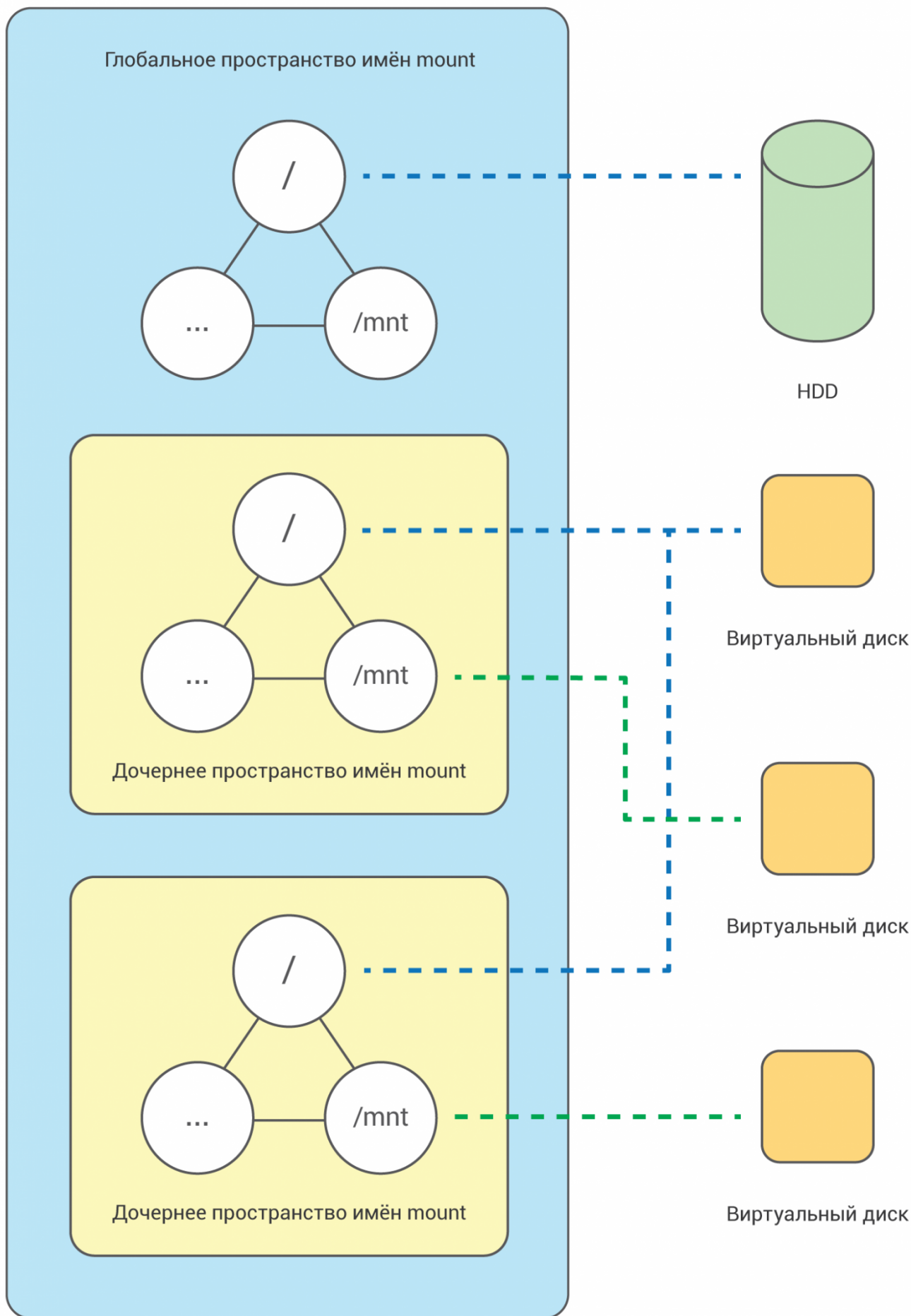
файловый дескриптор, который на это пространство имён ссылается. Он хранится в файле `/proc/<PID процесса>/ns/net`. Открыв этот файл, мы можем передать файловый дескриптор функции `setns()`.

Можно пойти и другим путём. При создании нового пространства имён с помощью команды `ip` создаётся файл в директории `/var/run/netns/` (см. в выводе трассировки выше). Чтобы получить файловый дескриптор, достаточно просто открыть этот файл.

Сетевое пространство имён нельзя удалить при помощи какого-либо системного вызова. Оно будет существовать, пока его использует хотя бы один процесс.

MOUNT: изоляция файловой системы

Об изоляции на уровне файловой системы мы уже упоминали выше, когда разбирали системный вызов `chroot()`. Мы отметили, что системный вызов `chroot()` не обеспечивает надёжной изоляции. С помощью же пространств имён MOUNT можно создавать полностью независимые файловые системы, ассоциируемые с различными процессами:



Для изоляции файловой системы используется системный вызов `clone()` с флагом

CLONE_NEWNS:

```
clone(child_fn, child_stack+1048576, CLONE_NEWPID | CLONE_NEWNET | CLONE_NEWNS | SI
```

Сначала дочерний процесс «видит» те же точки монтирования, что и родительский. Как только дочерний процесс перенесён в отдельное пространство имён, к нему можно примонтировать любую файловую систему, и это никак не затронет ни родительский процесс, ни другие пространства имён.

Другие пространства имён

Изолированный процесс также может быть помещён в другие пространства имён: UID, IPC и PTS. UID позволяет процессу получать привилегии `root` в пределах определённого пространства имён. С помощью пространства имён IPC можно изолировать ресурсы для коммуникации между процессами.

UTS используется для изоляции системных идентификаторов: имени узла (`nodename`) и имени домена (`domainname`), возвращаемых системным вызовом `uname()`. Рассмотрим ещё одну небольшую программу:

```
#define _GNU_SOURCE
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/utsname.h>
#include <sys/wait.h>
#include <unistd.h>

static char child_stack[1048576];

static void print_nodename() {
    struct utsname utsname;
    uname(&utsname);
    printf("%s\n", utsname.nodename);
}

static int child_fn() {
    printf("Новое имя: ");
    print_nodename();
}
```

```
printf("Имя будет изменено в новом пространстве имён!\n");
sethostname("NewOS", 6);

printf("Новое имя узла: ");
print_nodename();
return 0;
}

int main() {
    printf("Первоначальное имя узла: ");
    print_nodename();

    pid_t child_pid = clone(child_fn, child_stack+1048576, CLONE_NEWUTS | SIGCHLD, NU

    sleep(1);

    printf("Первоначальное имя узла: ");
    print_nodename();

    waitpid(child_pid, NULL, 0);

    return 0;
}
```

Вывод этой программы будет выглядеть так:

```
Первоначальное имя узла: lilah
Новое имя узла: lilah
Имя будет изменено в новом пространстве имён!
New UTS namespace nodename: NewOS
```

Как видим, функция `child_fn()` выводит имя узла, изменяет его, а затем выводит уже новое имя. Изменение происходит только внутри нового пространства имён.

Заключение

В этой статье мы в общих чертах рассмотрели, как работает механизм `namespaces`. Надеемся, она поможет вам лучше понять принципы работы контейнеров. По традиции приводим ссылки на интересные дополнительные материалы:

- цикл статей о механизме пространств имён на портале [LWN.net](https://lwn.net/);

- руководство по работе с namespaces;
- конспект подробной лекции о namespaces и cgroups.

Рассмотрение механизмов контейнеризации мы обязательно продолжим. В следующей публикации мы расскажем о механизме cgroups.

Если вы по тем или иным причинам не можете оставлять комментарии здесь – приглашаем в наш блог.

Теги: контейнеры, контейнеризация, linux, linux kernel, namespaces, пространства имён, селектел, selectel

Хабы: Блог компании Selectel

Редакторский дайджест

Присылаем лучшие статьи раз в месяц



Selectel

IT-инфраструктура для бизнеса

[ВКонтакте](#) [Telegram](#)



146

0

Карма

Рейтинг

Андрей Емельянов @AndreiYemeljanov

Пользователь



Комментировать

Публикации

ЛУЧШИЕ ЗА СУТКИ

ПОХОЖИЕ

**anton-artemenko**

вчера в 14:01

Идеальные паразиты человека и «тихая пандемия»: привет, ветрянка и герпес

**Простой**

13 мин



12K

Обзор

**+48**

55



47

21 час назад

Герои напильника и паяльника: итоги сезона DIY



8 мин



8.1K



Сезон DIY



Спецпроект

**+41**

42



6

**spiritus_sancti**

22 часа назад

RGB-усилители. Особенности, проблемы, выбор

**Простой**

6 мин



5.4K

Тutorial

**+39**

24



11

**ru_vds**

18 часов назад

Профилирование Python — почему и где тормозит ваш код

**Средний**

10 мин



3.2K

Tutorial

Перевод

**+31**

90



3

**ViktorSergeev**

15 часов назад

WebOne: даём жизнь старым браузерам



6 мин



3.1K

 +27 26 7

AnnaVIMorozova

18 часов назад

Как повысить эффективность коммуникаций в команде: учимся решать конфликты

 7 мин 2.1K +19 43 2

Yu-Leo

23 часа назад

Обзор электронной книги Meebook P10 Pro

 Простой 9 мин 6.1K[Обзор](#) +18 18 8

it_union

1 час назад

ЗАО Гейм Инсайт Труп

 Простой 4 мин 2.3K +15 6 5

fedorborovitsky

вчера в 14:13

Российский софт в идеальном шторме

 6 мин 7.6K[Мнение](#) +15 15 38

zatiм

46 минут назад

Видеокарта VGA для микроконтроллера

 Средний 17 мин 444[Тutorial](#)

Показать еще

ВАКАНСИИ КОМПАНИИ «SELECTEL»

- QA Team Lead (web)

Selectel · Санкт-Петербург
- Python engineer в команду Compute

Selectel · Можно удаленно
- Практикант в инженерно-технических отдел

Selectel · Санкт-Петербург
- QA Fullstack Engineer в команду разработки Выделенных серверов

Selectel · Можно удаленно
- Python-разработчик в команду Биллинговой платформы

Selectel · Можно удаленно
- Больше вакансий на Хабр Карьере

ИНФОРМАЦИЯ

| | |
|------------------|-------------------|
| Сайт | selectel.ru |
| Дата регистрации | 16 марта 2010 |
| Дата основания | 11 сентября 2008 |
| Численность | 501–1 000 человек |
| Местоположение | Россия |
| Представитель | Ульяна Малышева |

ССЫЛКИ

- Выделенный сервер от 26 рублей в день

selectel.ru
- Сервер для 3D - моделирования и рендеринга

selectel.ru

Физический сервер от 800 рублей в месяц
[selectel.ru](#)

Облачные серверы от 280 рублей в месяц
[selectel.ru](#)

FAQ
[slc.tl](#)


Реферальная программа
[slc.tl](#)

Telegram-канал о технологиях
[t.me](#)

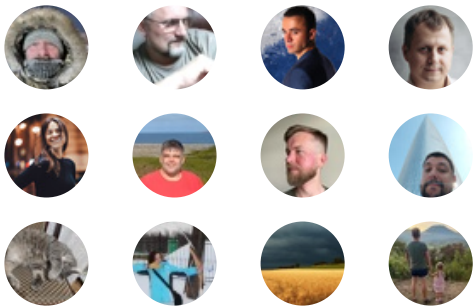
Telegram-канал про карьеру в IT
[t.me](#)

Вакансии
[slc.tl](#)

В КОНТАКТЕ

 **Selectel**

91 363 подписчика



[Подписаться на новости](#)

ВИДЖЕТ

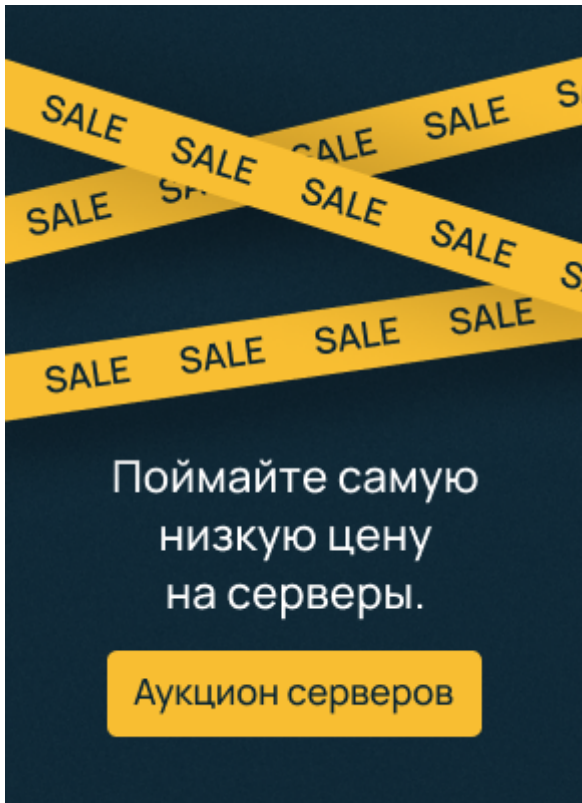
Selectel Career Wave

Стажировка в Backend
и DevOps для студентов
Санкт-Петербурга

Подай заявку до 10 сентября



ВИДЖЕТ



БЛОГ НА ХАБРЕ

5 часов назад

Всего два месяца — и новый релиз ядра Linux. Что появилось в ядре 6.5, что изменилось и что удалили. Новые возможности

 4.5K  4

вчера в 07:59

Гигабайт в мире мини-ПК: 96 ГБ ОЗУ и Intel Core i9-13900H в форм-факторе NUC

 8.5K  28

25 авг в 08:34

Новый ремонт Nintendo Switch Lite: как меня обманул продавец, но я все починил. Отвал процессора

 8.1K  45

24 авг в 18:07

MLOps от Gucci и оценка уровня Data Driven'ности в компании

 1.6K  0

23 авг в 17:31

Из Zero в Hero: как нетехническому специалисту работать со сложным продуктом

2.5K 1

Ваш аккаунт

Войти

Регистрация

Разделы

Статьи

Новости

Хабы

Компании

Авторы

Песочница

Информация

Устройство сайта

Для авторов

Для компаний

Документы

Соглашение

Конфиденциальность

Услуги

Корпоративный блог

Медийная реклама

Нативные проекты

Образовательные

программы

Стартапам

Спецпроекты



Настройка языка

Техническая поддержка

Вернуться на старую версию