GeekBrains

Механизмы пространства имен

Контейнеризация

Урок 1

Оглавление

Введение

Всем привет! На этом курсе мы познакомимся с контейнеризацией, изучим магию работы контейнеров и рассмотрим средства, с помощью которых и реализована работа в контейнерах.

На этом курсе

Мы изучим механизмы контейнеризации – выясним как устроена контейнеризация «под капотом» и для чего это нужно. Разберемся с системными вызовами, как и где они встроены в ядро системы, посмотрим на часть системных библиотек. Напишем свои собственные мини-программы, исследуем их работу и разберемся с результатами.

Docker: Dockerfile, Docker Compose, Docker Swarm. Разберем основные механизмы, определим когда и при каких условиях использовать тот или иной инструмент.

На этом уроке

1. Разберём виды пространств имен и определим: для чего они нужны.
2. Изучим механизм работы пространства имен.
3. Рассмотрим применение изоляции в системе Linux.
4. Изучим cgroups: рассмотрим появление и историю механизма.
5. Изучим архитектуру: из чего состоят cgroups.
6. Разберем примеры управления группами.
7. Рассмотрим недостатки механизма cgroups.

Вступление

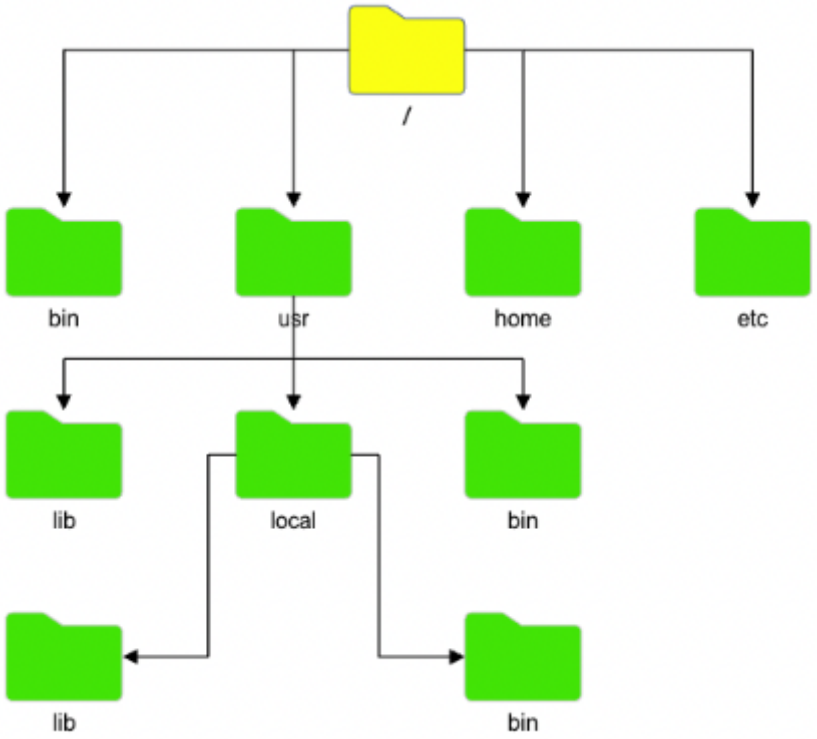
Ни для кого не секрет, что в последнее время происходит бурный рост популярности “контейнеризированных” решений как для Linux, так и для Windows-платформы. Однако, далеко не все знают, какие механизмы контейнеризации лежат в основе этой технологи. Об этом сегодня и пойдет речь. Этих механизмов два - namespaces и cgroups, и все остальные инструменты (Docker, LXC (Canonical) и так далее) основываются на них.

Файловая система

Давайте освежим знания о файловой системе так как сегодня один из больших аспектов - работа с ней и переназначение “корня” файловой системы. Абсолютно все объекты в ОС Linux - файлы. Делятся они лишь на типы:

- Обычные файлы - символьные и двоичные файлы: текстовые документы, исполняемые файлы, пользовательские данные и так далее.
- Каталоги (директории) - список ссылок на файлы либо на другие каталоги. Да-да! Это не та сущность, к которой мы привыкли в ОС Windows!
- Символьные ссылки - ссылки на другие файлы и каталоги по имени. Аналог в Windows - ярлык.
- Блочные устройства - интерфейсы, предназначенные для взаимодействия с аппаратной составляющей. В этом случае, при обращении к этому файлу, ядро ОС перенаправляет запрос далее - драйверу устройства.
- Сокеты и каналы - интерфейсы для взаимодействия с программной составляющей ОС.

Обновив информацию о типах файлов, можно обновить информацию о структуре файловой системы Linux в целом. На изображении приведен пример типичной структуры ФС Linux:



Корнем ФС является “корень” (от англ. root) и на изображении выделен “желтой папкой”. Давайте подробнее рассмотрим каждую директорию в ОС:

- **/bin** (binaries) – здесь содержатся исполняемые файлы самых необходимых утилит. **Важный момент:** эта папка может быть символьной ссылкой на /usr/bin.
- **/boot** – папка содержит файлы, которые необходимы для первичного этапа загрузки ОС Linux: загрузки ядра и его компоненты.
- **/dev** (devices) – содержит в себе блочные и символьные файлы устройств (диски, терминалы, клавиатуры, принтеры и так далее).
- **/etc** (etcetera) – хранит в себе набор конфигурационных файлов системы и прикладных программ.
- **/home** – в этой директории хранятся домашние каталоги пользователей для хранения «личных» файлов.
- **/lib** (libraries) – содержит в себе файлы библиотек, необходимых для работы утилит, системного и прикладного ПО. Также может быть символьной ссылкой на /usr/bin.
- **/mnt** (mount) – каталог, предназначенный для подключения файловых систем (съемных носителей и др.). Как принято называть - точка монтирования (от англ. mountpoint).

- **/opt** (optional) – каталог для различного рода дополнительных программ (проприетарных драйверов, агентов мониторинга и др.).
- **/proc** (process) – одна из важнейших директорий, содержащая в себе сущности-ссылки на файлы, содержащиеся в оперативной памяти, в которых содержится информация о выполняемых в системе процессах.
- **/root** – домашний каталог пользователя root (аналог /home для других пользователей).
- **/sbin** (system binaries) – содержит набор файлов системных утилит, которые необходимы для загрузки ядра ОС, резервного копирования и восстановления системы. Также может быть символьной ссылкой на /usr/sbin.
- **/sys** (system) – виртуальная файловая система sysfs, которая содержит информацию об аппаратном обеспечении (ЦПУ, ОЗУ, дисках, сетевых устройствах), драйверах, ядре системы и др.
- **/tmp** – каталог для различного рода временных файлов, обычно зачищается при каждой загрузке системы.
- **/usr** – ранее считалось, что это пользовательский каталог, который содержит каталоги исполняемых файлов и конфигурационных файлов. Сейчас же - просто каталог с исполняемыми и конфигурационными файлами.
- **/var** (variable) – содержит файлы, создаваемые или используемые различными программами (различные логи, очереди, идентификаторы процессов, базы данных и тд.).

Собственно, начать урок предлагается с изучения утилиты **chroot**, которая позволяет пользователю выполнить операцию изменения корневого каталога системы для запущенного процесса и всех его дочерних процессов. Приложение, запущенное в таком окружении, работает изолированно и не может получить доступ к остальным директориям файловой системы вне своего корневого каталога. Важный момент: для выполнения этой операции необходимо иметь права суперпользователя (sudo).

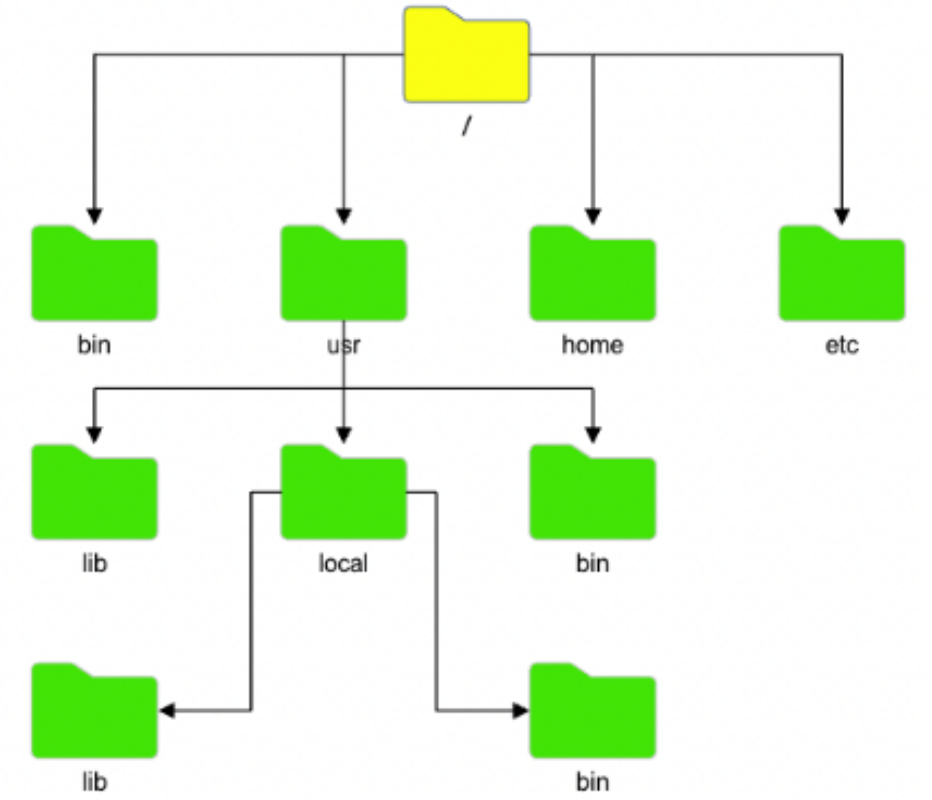
Историческая справка

Начать предлагается с самого начала так как идеи, которые лежат в основе механизма namespaces закладывались в архитектуру системы практически с самого начала разработки Linux. Системный вызов chroot был добавлен в механизм ядра еще в 1979 году с целью обеспечения изоляции программной оболочки и предоставления механизма для тестирования тех или иных компонент системы.

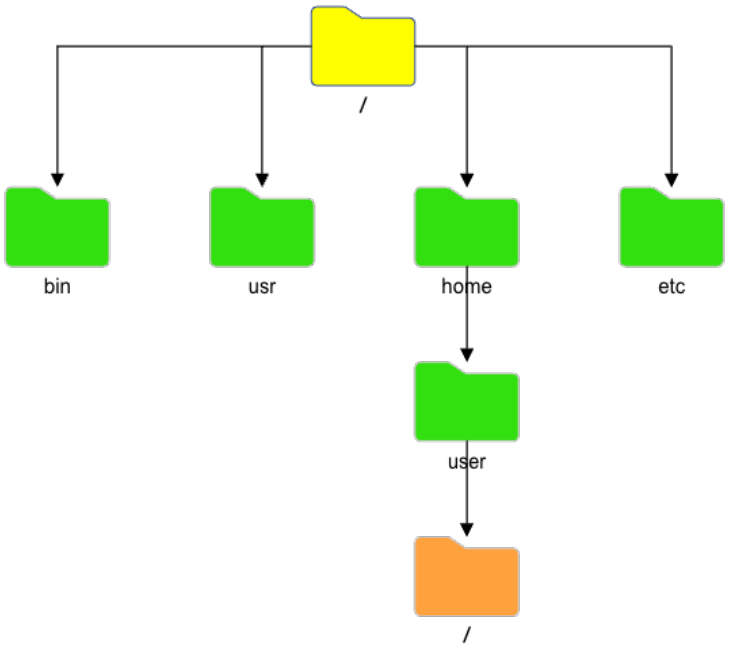
Первая изоляция: chroot

Дословно chroot - change root (от англ. “Изменить корень”). С помощью стандартного системного вызова chroot() и соответствующих аргументов можно изменить корневой каталог на введенный пользователем.

Как вам известно, файловая система UNIX-систем представляет из себя древовидную иерархию:



В корне иерархии - каталог “/” (root). Остальные же каталоги вложены в него. С помощью вызова функции chroot можно добавить второй корневой каталог, который с точки зрения пользовательского восприятия ничем не будет отличаться от первого. Формально, при добавлении второго корневого каталога, файловая структура будет выглядеть следующим образом:



В этом случае происходит разделение файловой системы на две составляющие, которые никоим образом не влияют друг на друга.

Для погружения в процедуру смены корневого каталога предлагается рассмотреть описание системного вызова в ОС BSD-Lite 4.4 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);
}
```

Данная функция довольно проста. В первой ее части происходит объявление самой функции: передаются входные параметры и далее - объявляются локально в виде структуры данных. Далее происходит регистрация данных, объявление кода

ошибки (не все же может пойти гладко). Затем происходит самое интересное - ближе к концу файла: директория указанная пользователем, становится корневой.

Аналогичный вызов имеется и в ядре Linux, однако реализация его куда более сложна:

```
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;
}
```

Практический разбор вызова chroot

Для лучшего понимания работы механизма, предлагается рассмотреть вызов chroot на практике. Для этого создадим каталог **testfolder** в домашнем каталоге пользователя и запустим команду:

```
mkdir testfolder # создаем новую директорию

chroot testfolder /bin/bash
```

Вторая команда содержит в себе 2 аргумента: первый - папка, которую мы хотим сделать корневой, второй - командный интерпретатор, который мы хотим использовать. Здесь можно использовать и другие варианты: /bin/sh, zsh и так далее.

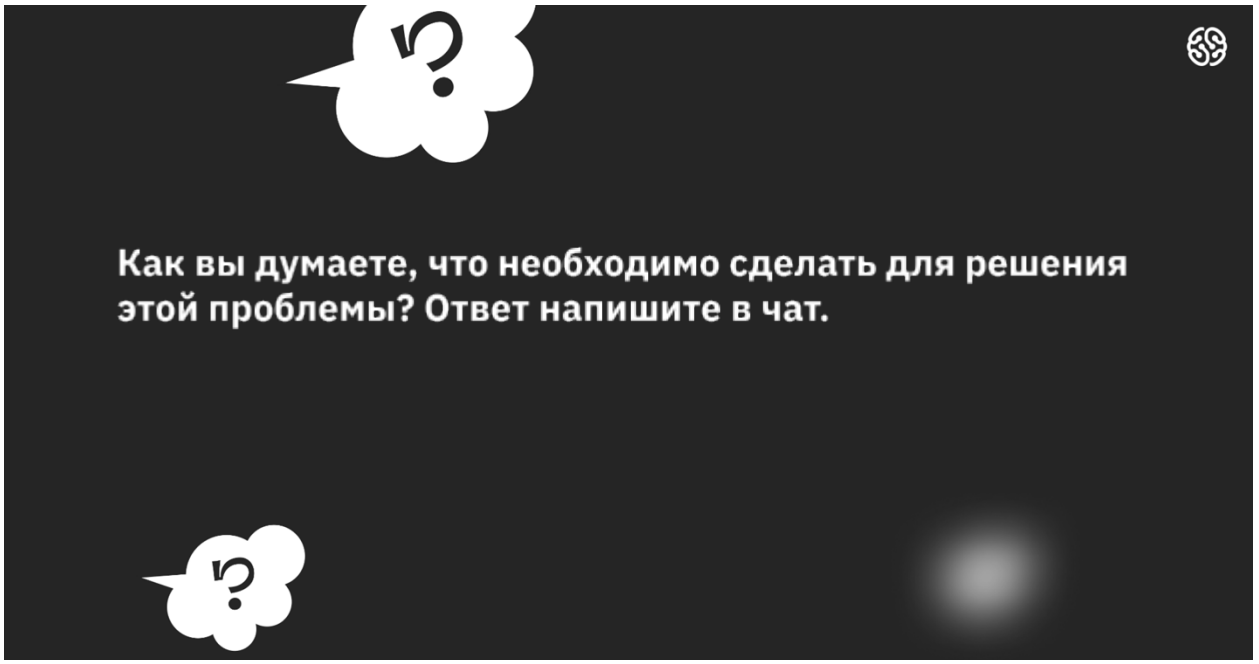
Однако, в результате выполнения этой команды мы получим сообщение об ошибке:

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

Скажу больше - ошибка более чем ожидаемая и она гласит: “Не найдена командная оболочка”. На этот момент и предлагается обратить внимание. В нашем случае, применяя команду, мы создаем абсолютно новую, изолированную файловую систему, не имеющей доступа к системной, которая содержится выше нового корня. В данном случае обращение к /bin/bash процесс будет выполнять через следующий путь: **/testfolder/bin/bash**. Для того, чтобы все запустилось корректно, необходимо несколько видоизменить набор команд: создать в “новом корне” папку bin, поместить туда исполняемый файл командного интерпретатора.

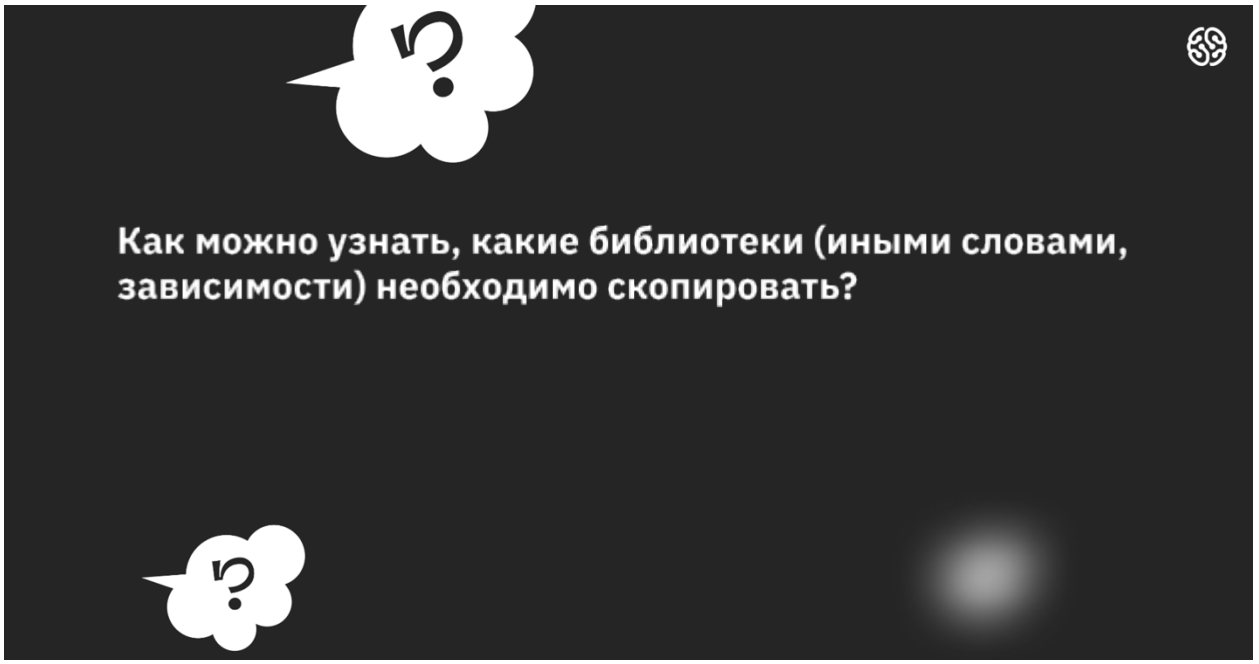
```
mkdir testfolder/bin

cp /bin/bash testfolder/bin

chroot testfolder

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

Казалось бы мы скопировали в изолированную файловую систему исполняемый файл. Однако, ошибка осталась. Несмотря на то, что текст ошибки выглядит точно также, расшифровка ее совсем иная. Теперь сам исполняемый файл присутствует и доступен. Проблема заключается в ином - не найдены необходимые библиотеки для запуска исполняемого файла.



Ответ: необходимо скопировать весь необходимый набор библиотек также в chroot-папку.

Следующий вопрос сложнее и точно не каждый знает ответа на вопрос.



Ответ: воспользоваться следующей командой:

```
    $ ldd /bin/bash
     linux-vdso.so.1 (0x00007ffee1fd2000)
     libtinfo.so.6 => /lib/x86_64-linux-gnu/libtinfo.so.6
(0x00007fdfe23e7000)
     libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
(0x00007fdfe21bf000)
```

```
 /lib64/ld-linux-x86-64.so.2 (0x00007fdfe2581000)
```

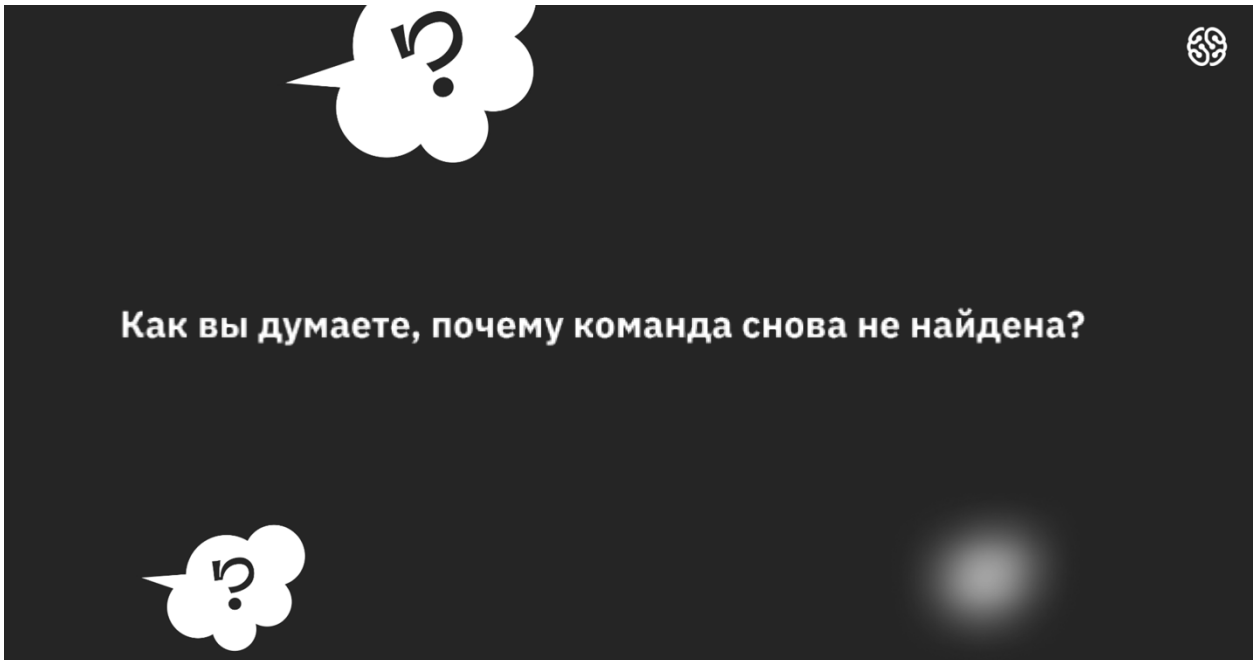
После того, как список необходимых зависимостей был получен, необходимо также скопировать их в новую файловую систему и попытаться повторить наш эксперимент:

```
$ cp /lib/x86_64-linux-gnu/libtinfo.so.6 testfolder/lib
$ cp /lib/x86_64-linux-gnu/libc.so.6 testfolder/lib
$ cp /lib64/ld-linux-x86-64.so.2 testfolder/lib64/
$ chroot testfolder
bash-5.1#
```

Ура! У нас все получилось. Как мы видим, теперь “приветствие” системы на ввод команды стало иным. Это и означает, что мы изолировали процесс. По сути, у нас запустилась изолированная оболочка командного интерпретатора **bash**, с корнем, отличным от остальной системы.

Теперь, когда у нас все получилось, давайте попробуем просмотреть текущую директорию с помощью команды **ls:**

```
bash-5.1# ls
bash: ls: command not found
```

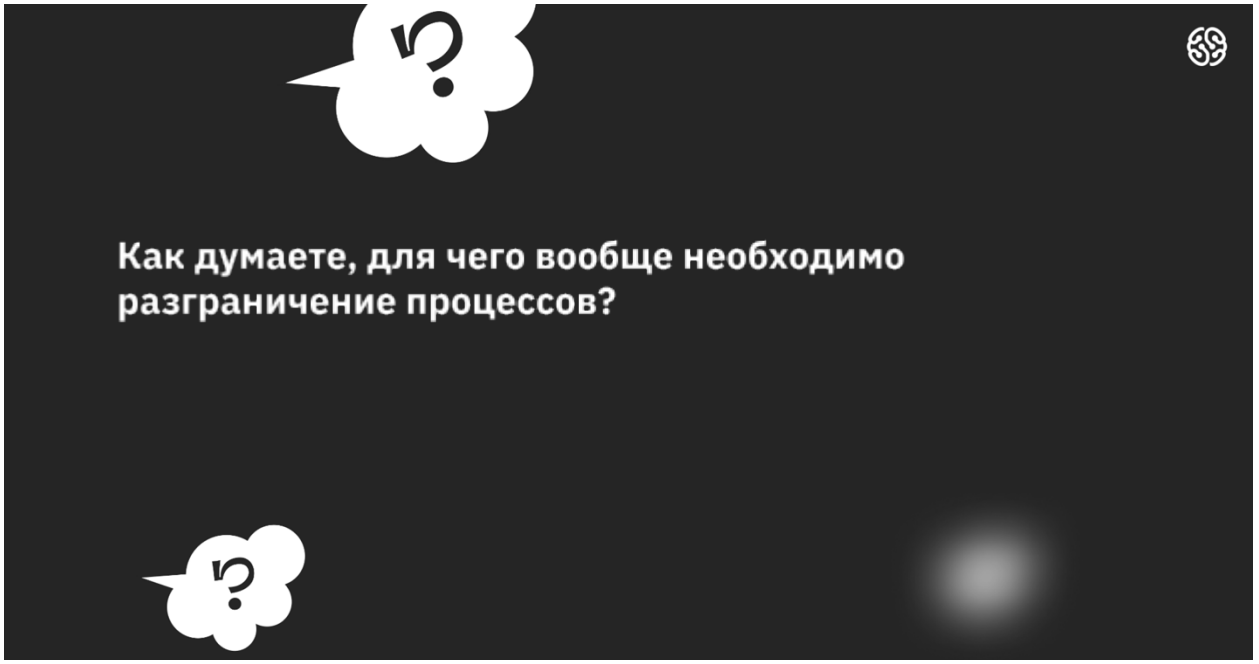


Ответ: как и ранее, необходимо скопировать все необходимые исполняемые файлы в созданную папку вместе с динамическими библиотеками.

Рассмотрев chroot на практике, можно определить серьезный недостаток использования chroot: при создании новой “корневой” директории, необходимо дублировать все используемые файлы, что, несомненно, повлечет за собой рост занимаемого дискового пространства.

Механизм пространств имен

Пространство имен (от англ. namespaces) - это абстракция в системе Linux, в которой находятся системные ресурсы. Тип ресурса зависит от типа пространства имен. Пространства имен - не какая-то дополнительная функция, которую можно установить или просто обойтись без нее. Эта сущность изначально предоставляется самим ядром ОС Linux и является необходимым компонентом, принимающим участие в процедуре запуска любого процесса в системе. В любой момент времени любой процесс может принадлежать только одному пространству имен каждого типа. Также namespace - это механизм, обеспечивающий изоляцию процессов друг от друга в UNIX-системах. Реализация данного механизма была начата с версии ядра 2.4.19 и продолжается по сей день.



Ответ: безопасность. Каждое отдельное пространство имен независимо. То есть мы можем делать абсолютно любые процедуры внутри этого пространства при этом не затрагивая остальные.

Пример: что будет если мы выполним в нашей системе следующую команду?

```
sudo rm -rf /
```

После выполнения этой команды система будет серьезно повреждена и не сможет даже произвести процедуру загрузки так как, выполнив эту команду, мы удалим ее корень, а вместе с ним система лишится и адресации файлов.

Если же эта команда будет выполнена внутри изолированного пространства имен, выйдет из строя лишь оно.

Давайте поговорим непосредственно о типах пространства имен. На данный момент имеется шесть типов namespace:

Пространство имен	**Что изолируется**
PID	Изоляция идентификатора процессов
Network	Сетевая изоляция (сеть, порты, стеки и так далее)

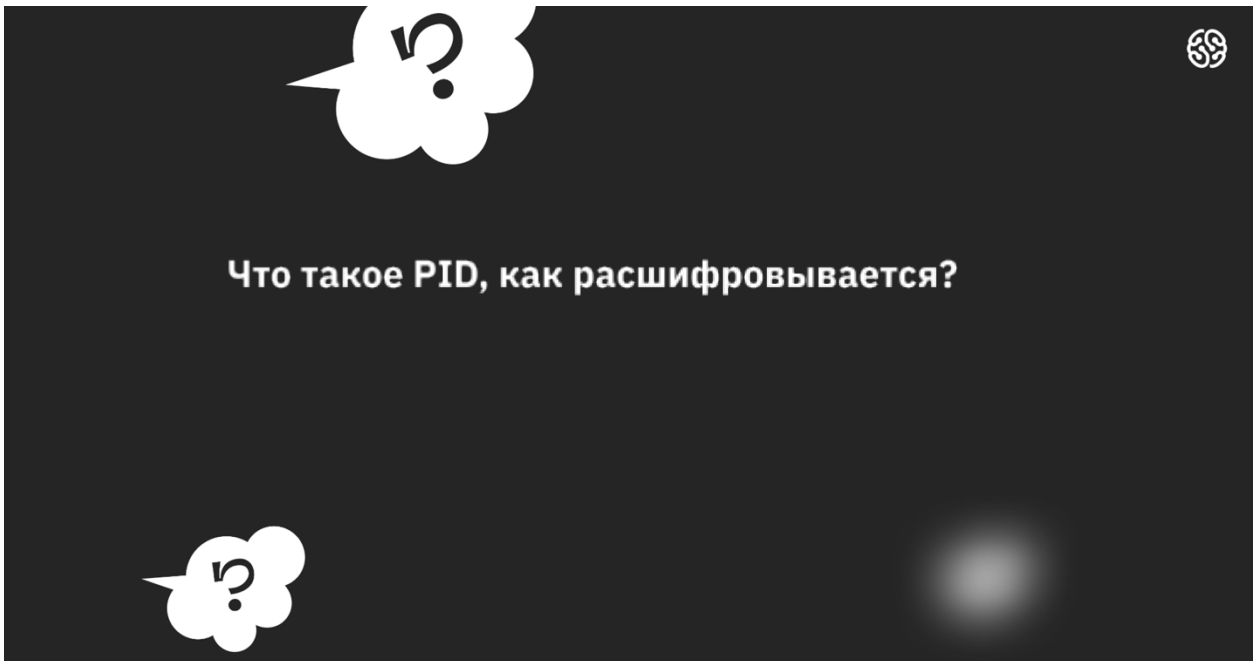
User	Изоляция пользователей и групп
Mount	Переопределение точек монтирования
IPC (Inter Process Communication)	Изоляция межпроцессного взаимодействия (в т.ч. очереди POSIX)
UTS (UNIX Time-Sharing)	Изоляция имени хоста и доменного имени

Давайте перед началом основной части дадим понятие нескольким сущностям:

Квота - количественные ограничения на что-либо.

Контейнер - это процесс в ОС Linux, с набором собственных пространств имен и квот. Также стоит отметить, что контейнер без квот вполне может принадлежать общему пространству имен, совместно используя ресурсы в них. Контейнеры — хорошая альтернатива аппаратной виртуализации. Они позволяют запускать приложения в изолированном окружении, но при этом потребляют намного меньше ресурсов.

Также, перед описанием пространств имен давайте вспомним:



Ответ: PID - это идентификационный номер процесса, который обычно используется большинством ядер операционной системы, таких как Linux, Unix, macOS и Windows. Это уникальный идентификационный номер, который автоматически присваивается каждому процессу, когда он создается в операционной системе.

Процесс – это исполняемый экземпляр программы.

Практика!

Каждый раз идентификатор процесса будет получать изменения ко всем процессам кроме init, поскольку init всегда является первым процессом в системе и является предком всех других процессов. Это PID – 1. Максимальное значение PID в 32-битных системах по умолчанию – 32 768.

Проверить этот факт довольно просто: достаточно лишь запустить следующую команду:

```
cat/proc/sys/kernel/pid_max
```

В 32-битных системах 32768 является максимальным значением, но мы можем установить любое значение до 2^22 (приблизительно 4 миллиона) в 64-битных системах.

Вы можете спросить, почему нам нужно такое количество PID? Потому что мы не можем повторно использовать PID сразу. Также во избежание возможных ошибок.

PID для запущенных процессов в системе можно найти с помощью, например, команды ps либо команды top:

ps – сообщает моментальный снимок текущих процессов.

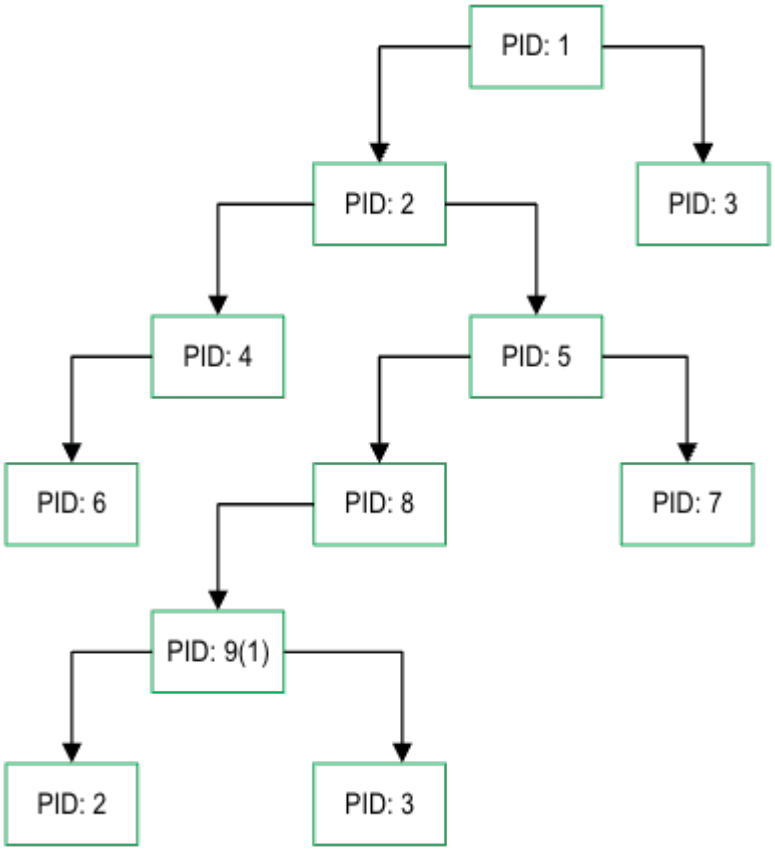
top выводит список запущенных процессов в реальном времени.

PID: изолируем идентификаторы процессов

Дерево процессов - иерархическая структура, подобная дереву каталогов файловой системы. Исторически была поддержка лишь одного древа процессов так как при создании ядра и системы в целом не было необходимости делить его: как минимум, не позволяли аппаратные мощности, да и программного обоснования этому также не было.

Однако, с появлением механизма пространства имен появилась возможность поддержки нескольких деревьев процесса, которые полностью были бы изолированы друг от друга. Как вы знаете, при загрузке Linux сначала запускается процесс с PID (идентификационный номер), равный 1. Он является корневым процессом системы и запускает остальные службы и процессы. Механизм пространства имен позволяет создавать отдельные ветвления в дереве с собственным (независимым) PID (который равен 1). Процессы, создающие такие ответвления являются частью основного древа процессов, однако, дочерний процесс в новом дереве будет корневым.

Важно отметить, что процессы нового дерева никак не могут взаимодействовать с родительским процессом и, следовательно, с основным деревом. Наглядно это выглядит следующим образом:



Важно отметить, что можно создавать вложенные пространства имен: основной процесс запускает дочерний в новом пространстве имен, а любой дочерний, в свою очередь, может породить новое пространство имен и так далее.

В данном случае, для создания пространства имен используется системный вызов **clone()** с дополнительным флагом **CLONE_NEWPID**. С помощью такого флага как раз и запускается новый процесс в дочернем пространстве имен, образуя новое поддерево.

Теория - это хорошо, но теперь давайте займемся практикой. Для проверки теории предлагается скомпилировать и запустить следующий программный код, написанный на языке C:

```
#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;
}
```

Описанное приложение позволяет вывести на экран PID процесса в общем древе, а также, чтобы убедиться, PID процесса в изолированном дереве.

Для компиляции потребуется компилятор gcc и непосредственно сама команда:

```
gcc example.c -o example.sh
```

После выполнения этой команды в той же папке появится исполняемый файл, готовый к выполнению. При запуске вы получите следующий вывод:

```
$ gcc example.c -o example.sh
$ ./example.sh
  clone() = 16158
  PID: 1
```

Пусть сама программа и крохотная, но в процессе ее выполнения произошли процессы, которые пользователю не виды. А именно: вызов функции clone() породил новый процесс, который клонировал текущий и начал его выполнение. Вместе с этим произошло отделение нового процесса от основного дерева, создав полностью изолированное дочернее. И самое главное: мы получили заветное значение: **PID: 1**! Это говорит о том, что произошла изоляция процесса и он находится в изолированном пространстве имен.

Следующий шаг - попытаться получить PID родительского процесса. Для этого в написанную программу необходимо внести крохотные изменения в функцию: static int child_fn()

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

Теперь нужно скомпилировать программу и запустить исполняемый файл. На выходе мы получим подобное:

```
$ ./example.sh
  clone() = 16170
  Parent PID: 0
```

“Как же так?!”, - можете спросить вы и усомнитесь в корректности предлагаемого вам кода. Однако, никакой ошибки в данном случае нет. Выданный результат говорит о том, что у задействованного нами процесса отсутствует родительский процесс.

Если же мы хотим получить PID родительского процесса, необходимо изменить вызов функции, убрав флаг **CLONE_NEWPID** из вызова функции **clone()**:

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

После этого предлагается снова скомпилировать программу и запустить. На выходе мы получим совершенно иной результат:

```
$ ./example.sh
  clone() = 16180
```

```
    Parent PID: 16179
```

А теперь давайте разбираться в произошедшем. Мы, поменяв вызов функции, сделали так, что **clone()** был исполнен практически также, как и **fork()**. В данном случае просто произошло создание нового процесса (родительского и дочернего). Да, между вызовами функций **clone()** и **fork()** есть существенное различие - **fork()** порождает дочерний процесс, который, в свою очередь, является копией родительского: вместе со всем наполнением для исполнения, выделением памяти, файловой подсистемой и так далее.

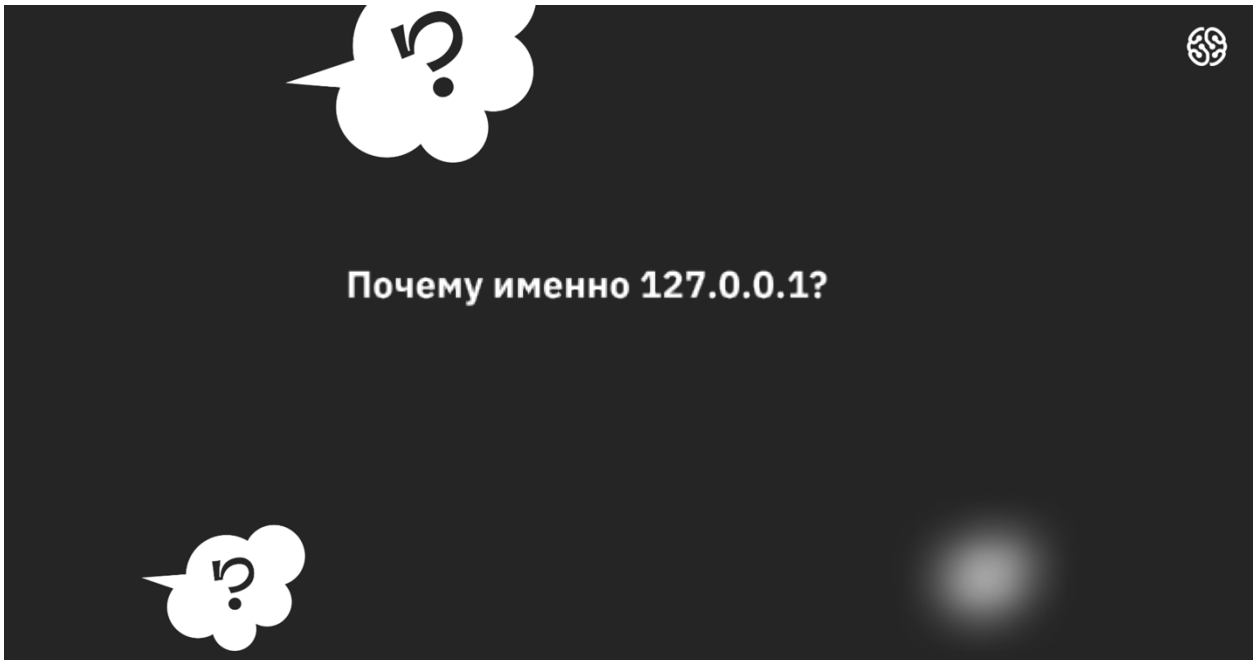
Сам же **clone()** не просто порождает копию процесса. Он позволяет разделить параметры и элементы выполнения между дочерними и родительскими процессами.

Таким образом, мы рассмотрели изоляцию на уровне процессов, однако, это лишь малая часть того, что мы рассмотрим далее. Важно отметить, что процесс, запущенный в своем собственном пространстве имен, может иметь доступ к абсолютно всем системным ресурсам. Если запущенный процесс будет слушать 80-й порт, то порт будет недоступен для остальных процессов так как занят текущим. Также использование только этого способа не позволяет изолировать и файловую систему. Для полной изоляции процесса предлагается изучить остальные пространства имен. Продолжить предлагается с сетевой изоляции, которая позволит освободить порты для остальных приложений и создать собственную, изолированную сетевую подсистему.

NET: Сетевая изоляция

Рассмотрев изоляцию PID, становится понятно, что контейнеры изолируются друг от друга различными типами изоляции и какого-либо одного из них определенно недостаточно. С помощью сетевого пространства имен можно не только изолировать процессы, но и создавать им собственные интерфейсы. Также при подобном типе изоляции, у каждого отдельного namespace создается свой виртуальный loopback-интерфейс.

Loopback-интерфейс - адрес возвратной петли. Это канал коммуникации с одной конечной точкой. Любые сообщения, посылаемые на этот канал, немедленно принимаются тем же самым каналом. Любые сообщения, которые отправляются с этого интерфейса, но у которых адрес не Loopback Interface, отбрасываются. В компьютерах таким адресом loopback interface является адрес 127.0.0.1, он фиксированный и изменению не подлежит. На Unix-like системах loopback interface называется lo или lo0.



Ответ: ответ на этот вопрос заключается в том факте, что к 1981 году 0 и 127 были единственными зарезервированными сетями класса A. Поскольку 0 использовался для указания на конкретный хост, то 127, последний номер сети, был оставлен для петлевого IP-адреса или локального хоста.

Нет реальной разницы между 127.0.0.1 и localhost. Некоторым программам может быть использовано цифровое обозначение, некоторым - буквенное, но обе они указывают на одно и то же местоположение: ваш компьютер.

В компьютерных сетях стандартное, официально зарезервированное доменное имя для частных IP-адресов (в диапазоне 127.0.0.1 — 127.255.255.254).

Для сети, состоящей только из одного компьютера, как правило, используется всего один адрес — 127.0.0.1, который устанавливается на специальный сетевой интерфейс «внутренней петли» (англ. loopback) в сетевом протоколе TCP/IP.

В этом случае мы все также будем использовать системный вызов **clone()**, однако, флаг вызова будет отличаться. Теперь нас интересует **CLONE_NEWNET** (в то время, как при PID-изоляции мы использовали **CLONE_NEWPID**).

Как и в прошлый раз для проверки наших теорий предлагается использовать крохотную программу:

```
#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("New `net` Namespace:\n");
  system("ip link");
  printf("\n\n");
  return 0;
}

int main() {
  printf("Original `net` Namespace:\n");
  system("ip link");
  printf("\n\n");
```

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

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

Скомпилировав и запустив код, можно получить следующий вывод:

```
Original `net` Namespace:
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state
UNKNOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: ens160: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
mq state UP mode DEFAULT group default qlen 1000
    link/ether 00:50:56:bc:aa:18 brd ff:ff:ff:ff:ff:ff
    altname enp3s0

New `net` Namespace:
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode
DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

Как можно видеть, в выводе нашего приложения отображено, что на хосте имеется 2 устройства: ens160 и loopback-интерфейс. В новом же пространстве появился еще один loopback-интерфейс, полностью изолированный от хостового. Проверить принадлежность довольно просто: необходимо лишь запустить утилиту **ip**, которая и покажет наличие 2 интерфейсов у хостовой машины. Также стоит отметить, что этот интерфейс не будет доступен в новом сетевом пространстве имен, так как оно полностью изолировано. Та же участь и у loopback-интерфейса хоста: доступа в новый namespace он не имеет.

В данном случае очевидно, что для корректной работы приложений в новом пространстве недостаточно одного лишь loopback-интерфейса. Необходимо

настроить необходимые нам виртуальные сетевые интерфейсы, которые будет охватывать одно или сразу несколько пространств имен. А уже после этого - создавать мосты Ethernet и даже производить маршрутизацию пакетов между пространствами имен.

Напоследок, чтобы это все работало в глобальном пространстве имен, необходимо запустить и процесс маршрутизации, чтобы можно было получать трафик от физического интерфейса и направлять его через соответствующие виртуальные интерфейсы в необходимые дочерние пространства имен.

Описав этот механизм, думаю, становится понятнее: почему так популярен **Docker** и похожие инструменты. Они просто делают все эти процедуры за нас!

Более подробно процесс работы можно отследить с помощью утилиты **strace <command_name>**. Прежде чем мы поговорим о выводе и использовании, стоит дать маленькое отступление о том, что из себя представляет утилита **strace** - это приложение, отслеживающее системные вызовы, которые представляют собой некий механизм трансляции, обеспечивающий виртуальный интерфейс между, непосредственно, процессом и ядром ОС. Эти вызовы можно перехватить и прочитать. В нашем случае мы этим и займемся для достижения лучшего понимания поведения процессов, задействованных в создании пространства имен.

Итак, вывод **strace** будет довольно длинным. Нас интересует окончание вывода:

```
    clone3({flags=CLONE_VM|CLONE_VFORK,      exit_signal=SIGCHLD,
stack=0x7efe6bf1b000, stack_size=0x9000}, 88) = 19393

    munmap(0x7efe6bf1b000, 36864)           = 0

    rt_sigprocmask(SIG_SETMASK, [CHLD], NULL, 8) = 0

    wait4(19393, 1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc
noqueue state UNKNOWN mode DEFAULT group default qlen 1000

        link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00

    2: ens160: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
mq state UP mode DEFAULT group default qlen 1000

        link/ether 00:50:56:bc:aa:18 brd ff:ff:ff:ff:ff:ff

        altname enp3s0

    3: ens192: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
mq state UP mode DEFAULT group default qlen 1000
```

```
    link/ether 00:50:56:bc:41:7d brd ff:ff:ff:ff:ff:ff
    altname enp11s0
4: ens224: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
mq state UP mode DEFAULT group default qlen 1000
    link/ether 00:50:56:bc:20:f2 brd ff:ff:ff:ff:ff:ff
    altname enp19s0
5: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500
qdisc noqueue state DOWN mode DEFAULT group default
    link/ether 02:42:c4:61:3c:75 brd ff:ff:ff:ff:ff:ff
[{WIFEXITED(s) && WEXITSTATUS(s) == 0}], 0, NULL) = 19393
rt_sigaction(SIGINT, {sa_handler=SIG_DFL, sa_mask=[],
sa_flags=SA_RESTORER, sa_restorer=0x7efe6bf69520}, NULL, 8) = 0
rt_sigaction(SIGQUIT, {sa_handler=SIG_DFL, sa_mask=[],
sa_flags=SA_RESTORER, sa_restorer=0x7efe6bf69520}, NULL, 8) = 0
rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
--- SIGCHLD {si_signo=SIGCHLD, si_code=CLD_EXITED,
si_pid=19393, si_uid=0, si_status=0, si_utime=0, si_stime=0} ---
write(1, "\n", 1
) = 1
write(1, "\n", 1
) = 1
clone(child_stack=0x564aa6e1f030,
flags=CLONE_NEWPID|CLONE_NEWNET|SIGCHLD) = 19395
wait4(19395, New `net` Namespace:
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode
DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00



NULL, 0, NULL) = 19395
```

```
    --- SIGCHLD {si_signo=SIGCHLD, si_code=CLD_EXITED,
si_pid=19395, si_uid=0, si_status=0, si_utime=0, si_stime=0} ---
    exit_group(0)                           = ?
    +++ exited with 0 +++
```

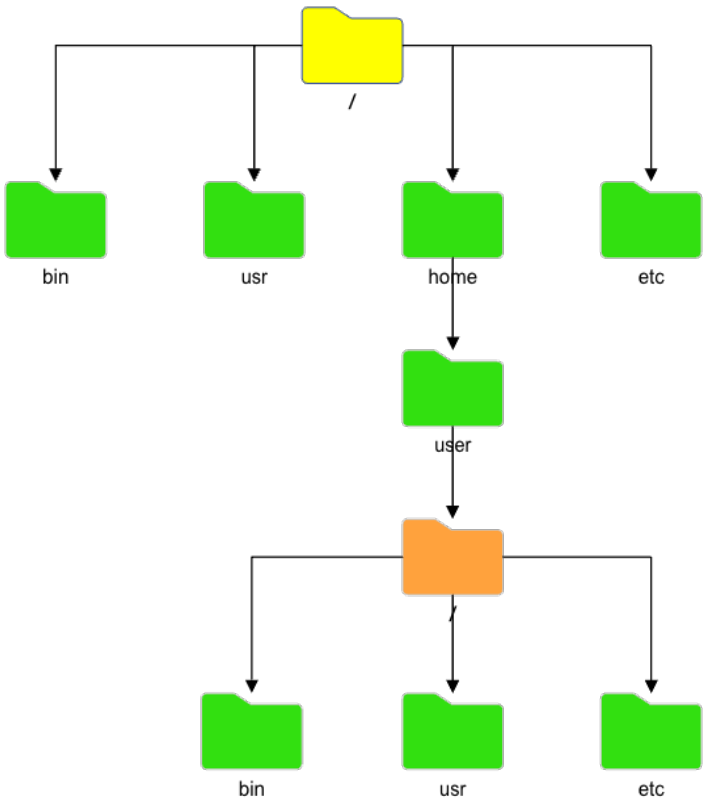
В выводе четко выражено использование функции **clone()**. В первом случае происходит передача аргументов, для создания дочернего процесса, а во втором вызове - создание подпространства с интересующими нас флагами: **CLONE_NEWPID** и **CLONE_NEWNET**.

Важное замечание: созданное сетевое пространство имен просто так нельзя удалить с помощью системного вызова. Любое пространство имен продолжает существовать до тех пор, пока его использует хотя бы один процесс.

MOUNT: Изолируем файловую систему целиком

На этом месте вы можете сказать: “Но мы же уже ранее занимались изоляцией файловой системы путем создания новой корневой директории с помощью вызова chroot!” И вы будете правы! Однако, изоляция с помощью вызова этой функции не обеспечивает достаточно надежную изоляцию.

Теперь же с помощью пространства имен MOUNT будет рассказано о том, как создавать целиком независимые файловые системы, которые можно ассоциировать с различными процессами. Выглядеть это будет следующим образом:



Данная структура также включает в себя информацию о том, какие разделы диска монтируются, куда они смонтированы, какие права выданы разделам (только чтение, чтение и запись) и так далее. Как уже говорилось ранее, с пространством имен MOUNT можно клонировать структуру данных, чтобы процессы, находящиеся в разных пространствах имен, могли изменять только собственные точки монтирования и, в принципе, данные, не влияя на процесс работы друг друга.

Важно отметить, что эффект от создания отдельного пространства имен монтирования практически ничем не отличается от уже рассмотренного нами использования утилиты chroot. Повторюсь, chroot() хорош и его можно использовать, но он не обеспечивает полной изоляции - это первый момент. Второй - эффект от работы этой функции ограничивается лишь монтированием корневой точки. Создание же отдельного пространства имен позволяет каждому из изолированных процессов иметь свое собственное представление о структуре точек монтирования всей системы. Это позволяет не только иметь отдельный корень монтирования для каждого процесса, но и другие собственные точки монтирования. Они могут быть уникальными для каждого процесса.

При должном использовании, можно корректным образом изолировать каждый процесс, целиком защитив базовую систему. Также с этим механизмом мы предотвращаем раскрытие какой-либо информации между процессами и базовой подсистемой (установленной ОС).

В данном случае, для изоляции ФС будет использован уже известный нам системный вызов clone(), но с флагом CLONE_NEWNS. Теперь же вызов будет выглядеть следующим образом:

```
    clone(child_fn,     child_stack+1048576,     CLONE_NEWPID     |
CLONE_NEWNET | CLONE_NEWNS | SIGCHLD, NULL)
```

В этом случае дочерний процесс при создании будет видеть все те же точки монтирования, что и родительский (т.е. всю файловую систему). Но, как только в процессе вызова процесс перенесется в отдельное пространство имен, к нему будет монтироваться только его, локальная ФС, не имеющая ничего общего с основной, то есть полностью изолированная. Также процесс сможет самостоятельно монтировать либо размонтировать любые конечные точки и это изменение не повлияет ни на пространство имен родительского процесса, ни на базовую подсистему.

Давайте погрузимся более детально. Если родительский процесс имеет одну точку монтирования в определенном разделе диска, изолированный (дочерний) также будет иметь эту точку монтирования в момент инициализации. К сожалению, это происходит довольно быстро и отследить опытным путем не получится. В момент, когда инициализация пройдена, у дочернего процесса появляется собственное (изолированное) пространство.

Остальные пространства имен

Глобальные и самые важные пространства имен мы рассмотрели. Теперь стоит поговорить об остальных, которые выполняют меньшие задачи, но все также важны для работы.

UID, например, позволяет каждому процессу получить привилегии суперпользователя (root) в пределах определенного пространства имен. В этом случае можно отдать процессу не глобальный неограниченный доступ ко всему, а изолированный.

UTS - его мы можем использовать для изоляции различных системных идентификаторов, таких как: имя узла (hostname), доменное имя (domainname). Все они возвращаются встроенным системным вызовом **uname()**.

uname - это инструмент, используемый для определения архитектуры процессора, hostname, domainname, версии ядра системы и многих других параметров. Вызывать его необходимо с набором параметров:

-s, (–kernel-name) – выводит имя ядра.

-n, (–nodename) – выводит имя узла системы (имя хоста). Это имя, которое система использует при общении по сети. При использовании с опцией -n uname показывает тот же вывод, что и команда hostname.

-r, (–kernel-release) – выводит выпуск ядра.

-v, (–kernel-version) – выводит версию ядра.

-m, (–machine) – выводит название аппаратного имени.

-p, (–processor) – выводит архитектуру процессора.

-i, (–hardware-platform) – выводит информацию об аппаратной платформе.

-o, (–operating-system) – распечатает название операционной системы. В системах Linux это «GNU/Linux».

-a, (–all) – при использовании опции -a uname ведет себя так же, как если бы были заданы опции -snrvmo.

Проверить работу функции можно довольно просто. Давайте узнаем имя ядра нашей системы:

```
uname -s
```

В ответ мы получим название ядра установленной системы.

Давайте теперь узнаем ее версию:

```
uname -v
```

Теперь мы видим не только название, но и полную версию ядра, на котором работает наша система.

Окей, с простыми вещами мы разобрались. Давайте снова проверим теорию с помощью программы, написанной на языке C. Код ее выглядит следующим образом:

```
#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("New UTS namespace nodename: ");
  print_nodename();

  printf("Changing nodename inside new UTS namespace\n");
  sethostname("GeekBrains", 6);

  printf("New UTS namespace nodename: ");
  print_nodename();
  return 0;
}

int main() {
  printf("Original UTS namespace nodename: ");
```

```
    print_nodename();

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

    sleep(1);

    printf("Original UTS namespace nodename: ");
    print_nodename();

    waitpid(child_pid, NULL, 0);

    return 0;
}
```

Как и ранее в первом блоке, происходит подключение необходимых для работы библиотек. Далее идет функция, которая выведет информацию о новом пространстве имен и завершается код программы основным блоком (int main), в котором описывается создание нового пространства имен:

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

И распечатка всей полученной информации. Вывод программы, к слову, выглядит следующим образом:

```
# ./example.sh
Original UTS namespace nodename: bastion
New UTS namespace nodename: bastion
Changing nodename inside new UTS namespace
New UTS namespace nodename: TEST
Original UTS namespace nodename: bastion
```

Резюмируем: здесь функция **child_fn()** выводит на экран сначала реальное имя узла, затем изменяет его для дочернего пространства имен, а только потом выводит новое имя, которое доступно в новом пространстве имен.

Заключение

Итак, на нашем первом уроке мы рассмотрели механизм работы пространств имен (они же - namespaces). Теперь вы знаете как можно изолированно запустить приложение, как изменить корень файловой системы, а также для чего это нужно. В дальнейшем мы познакомимся еще с рядом встроенных в систему механизмов для более точного понимания механизма работы контейнеров.

Важное замечание!

Для работы на семинаре необходимо иметь чистую виртуальную машину с предустановленной Операционной Системой Linux Ubuntu Server версии не ниже 20.04. В противном случае у вас не будет возможности в полной мере выполнять все задания, заготовленные для практики.

Скачать образ ОС можно по ссылке: https://ubuntu.com/download/server

Термины, используемые в лекции

LXC - это интерфейс пользовательского пространства для функций сдерживания ядра Linux. Благодаря мощному API и простым инструментам пользователи Linux могут легко создавать системные контейнеры или контейнеры приложений и управлять ими.

PID - (англ. Process IDentifier, PID) — уникальный номер (идентификатор) процесса в многозадачной операционной системе (ОС). В ОС семейства Windows PID хранится в переменной целочисленного типа. В ОС, соответствующих стандарту POSIX, тип переменной, хранящей PID, определяется каждой ОС индивидуально. Например, в ОС Linux PID хранится в переменной целочисленного типа (int).

IPC - (англ. inter-process communication, IPC) — обмен данными между потоками одного или разных процессов. Реализуется посредством механизмов, предоставляемых ядром ОС или процессом, использующим механизмы ОС и реализующим новые возможности IPC. Может осуществляться как на одном компьютере, так и между несколькими компьютерами сети.

UTF-8 (от англ. Unicode Transformation Format, 8-bit — «формат преобразования Юникода, 8-бит») — распространённый стандарт кодирования символов, позволяющий более компактно хранить и передавать символы Юникода, используя переменное количество байт (от 1 до 4), и обеспечивающий полную обратную совместимость с 7-битной кодировкой ASCII. Кодировка UTF-8 сейчас является доминирующей в веб-пространстве. Она также нашла широкое применение в UNIX-подобных операционных системах.

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

Операционная система, сокр. ОС (англ. operating system, OS) — комплекс управляющих и обрабатывающих программ, которые, с одной стороны, выступают как интерфейс между устройствами вычислительной системы и прикладными программами, а с другой стороны — предназначены для управления устройствами, управления вычислительными процессами, эффективного распределения вычислительных ресурсов между вычислительными процессами и организации надёжных вычислений.