Тема 5 - Изоляция процессов ОС Linux

Table of Contents

Безопасность операционных систем

Осень 2025 г.

Веричев Александр Владимирович
@xanchelavederiver alexandervерichev@gmail.com

1. Механизм chroot в Unix/Linux

Механизм `chroot` (*change root*) — один из самых ранних способов изоляции процессов в Unix-подобных операционных системах. Он позволяет изменить корневой каталог (`/`) для процесса и всех его потомков, создавая *иллюзию* работы процессов в отдельной файловой системы [1][2].

В настоящем разделе кратко рассмотрена история, назначение и внутренние механизмы работы `chroot`, практические примеры использования, а также приведено сравнение с современными контейнерными технологиями.

1.1. Краткий исторический обзор

`chroot` был представлен в **Unix Version** 7 в 1979 году, предположительно Биллом Джой (Bill Joy), одним из разработчиков BSD.

Изначально `chroot` создавался:

- как инструмент тестирования,
- для отладки программ в альтернативной файловой иерархии.

Изначально `chroot` **не задумывался** как механизм безопасности.

Со временем `chroot` был перенят BSD, System V и Linux и стал использоваться для запуска сетевых сервисов.

`chroot` оказал влияние на появление более сложных механизмов изоляции:

- FreeBSD Jails;
- Solaris Zones;
- Linux Containers (LXC);
- Docker.

1.2. Назначение chroot

`chroot` изменяет корневой каталог процесса, тем самым ограничивая доступ процесса к файловой системе.

Основная идея: процесс "видит" только ту часть файловой системы, которая находится внутри нового корня.

Table 1. Назначение chroot

Проблема	Как chroot помогает
Случайное повреждение системы	Процесс не имеет доступа к реальному `/`

Проблема	Как chroot помогает
Тестирование ПО	Использование альтернативных `/lib`, `/etc`
Ограничение ущерба	Компрометация сервиса ограничена файловой системой
Восстановление системы	Работа с установленной ОС из Live-среды

`chroot` **не является** полноценной границей безопасности (*chroot is not a security boundary*) так как:

- не создает новые пространства имён (*namespace*, см. Механизм пространства имён Linux namespaces) процессов;
- не ограничивает системные вызовы;
- не фильтрует сетевой трафик;
- не способен пользователю `root` выйти из `chroot jail` при неправильной настройке.

Механизм `chroot` **не изолирует**:

- сеть;
- процессы;
- пользователей;
- ядро;
- IPC;
- системные вызовы.

1.3. Механизм работы chroot

1.3.1. Системный вызов chroot()

`chroot jail` создаётся посредством команды chroot(1) или системного вызова chroot(2):

```
#include <unistd.h>

int chroot(const char *path);
```

Системный вызов chroot(2):

- доступен только привилегированным процессам (`root`);
- изменяет указатель корневого каталога процесса;
- влияет только на текущий процесс и его потомков.

1.3.2. Разрешение путей

Внутри `chroot jail` выполняется трансляция (разрешение) путей:

1. В абсолютных путях переписываются префиксы
   - до `chroot`

     ```
     /etc/passwd → /etc/passwd
     ```

   - после `chroot("/newroot")`

     ```
     /etc/passwd → /newroot/etc/passwd
     ```

2. Относительные пути продолжают разрешаться относительно текущего каталога.

1.3.3. Пример создания chroot jail

В качестве примера использования `chroot` создадим небольшую песочницу с корнем в директории `/tmp/chroot-test`.

1. Пробуем запустить `/bin/bash` с корнем на `/tmp/chroot-test`

```
$ mkdir /tmp/chroot-test
$ sudo chroot /tmp/chroot-test /bin/bash
chroot: failed to run command ‘/bin/bash’: No such file or directory
```

Ошибка: не найден исполняемый файл командной оболочки.

Так как `chroot` создаёт новую, изолированную файловую систему, которая не имеет доступа к основной, все файлы оригинальной системы оказываются недоступны - в том числе и `/bin/bash`.

2. Копируем `/bin/bash` в корень `/tmp/chroot-test` до запуска `chroot`

```
$ mkdir /tmp/chroot-test/bin
$ cp /bin/bash /tmp/chroot-test/bin
$ sudo chroot /tmp/chroot-test /bin/bash
chroot: failed to run command ‘/bin/bash’: No such file or directory
```

Опять ошибка! Однако, несмотря на идентичное сообщение, совсем не такая, как в прошлый раз.

В первой попытке не был найден сам исполняемый файл `/bin/bash`, теперь же не найдены динамически подключаемые библиотеки (`*.so`), с которыми скомпилирован интерпретатор.

3. Копируем все зависимости в корень `/tmp/chroot-test` до запуска `chroot`

Посмотреть, какие именно динамические библиотеки требуется скопировать, можно с

помощью `ldd`:

```
$ ldd /bin/bash
        linux-vdso.so.1 (0x000079298c924000)
        libtinfo.so.6 => /lib/x86_64-linux-gnu/libtinfo.so.6 (0x000079298c76e000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x000079298c400000)
        /lib64/ld-linux-x86-64.so.2 (0x000079298c926000)
```

Копируем зависимости и попробуем:

```
$ mkdir /tmp/chroot-test/lib /tmp/chroot-test/lib64
$ cp /lib/x86_64-linux-gnu/libtinfo.so.6 /tmp/chroot-test/lib/
$ cp /lib/x86_64-linux-gnu/libc.so.6 /tmp/chroot-test/lib
$ cp /lib64/ld-linux-x86-64.so.2 /tmp/chroot-test/lib64/
$ sudo chroot /tmp/chroot-test /bin/bash
bash-5.2#
```

Удобно настроить приглашение командной оболочки (*prompt*), дабы отличать среду `chroot jail` от основной, для этого в `chroot jail` выполнить:

```
export PS1="(chroot) $PS1"
```

Продолжая текущий пример:

```
$ sudo chroot /tmp/chroot-test /bin/bash
bash-5.2# export PS1="(chroot) $PS1"
(chroot) bash-5.2#
```

1.3.4. Пример корректного создания chroot jail

Для полноценного функционирования ОС в `chroot jail` необходимо выполнить несколько подготовительных операций [2].

Конкретные шаги могут отличаться для различных дистрибутивов. В настоящем разделе показан пример подготовки окружения в Ubuntu 24.04.

1. Подготавливаем окружение

```
$ mkdir /tmp/chroot-test2
$ mkdir /tmp/chroot-test2/{proc,sys,dev,run}
```

```
$ sudo mount -t proc proc /tmp/chroot-test2/proc/
$ sudo mount -t sysfs /sys /tmp/chroot-test2/sys/
$ sudo mount --rbind /dev /tmp/chroot-test2/dev/
```

```
$ sudo mount --rbind /run /tmp/chroot-test2/run/
```

2. Копируем необходимые исполняемые файлы и зависимости и работаем

Далее требуется либо скопировать исполняемые файлы и их зависимости, как показано в предыдущем примере.

Альтернативный вариант: смонтировать директории `/bin`, `/lib`, `/lib32`, `/lib64` и т.п. в одноимённые поддиректории корня:

```
$ mkdir /tmp/chroot-test2/{bin,lib,lib64}
$ sudo mount --rbind /bin /tmp/chroot-test2/bin
$ sudo mount --rbind /lib /tmp/chroot-test2/lib
$ sudo mount --rbind /lib64 /tmp/chroot-test2/lib64
```

Далее работаем в окружении:

```
$ sudo chroot /tmp/chroot-test2 /bin/bash
bash-5.2# export PS1="(chroot) $PS1"
(chroot) bash-5.2#
```

3. Зачищаем окружение

```
$ sudo umount /tmp/chroot-test2/lib64
$ sudo umount /tmp/chroot-test2/lib
$ sudo umount /tmp/chroot-test2/bin
```

```
$ sudo umount /tmp/chroot-test2/proc
$ sudo umount /tmp/chroot-test2/sys
$ sudo umount -l /tmp/chroot-test2/run
$ sudo umount -l /tmp/chroot-test2/dev
```

Чаще всего отмонтировать `dev/` и `run/` без флага `-l` (*lazy*) не получится, поэтому рекомендуется сразу по завершении работы с `chroot jail` выполнить перезагрузку системы.

1.4. Практические примеры использования механизма chroot

1.4.1. Пример 1: изоляция приложения

Одно из основных назначений `chroot` - запуск приложения в изолированном окружении.

Последовательность действий аналогична примерам, показанным в

примерах раздела Механизм работы chroot.

Для этого сначала требуется создать минимальное окружение:

```
$ mkdir -p /chroot/{bin,lib,lib64,...}
$ cp /bin/bash /chroot/bin/
...
```

Далее выполнить копирование исполняемых файлов и их зависимостей (библиотек `.so`):

```
$ ldd my-program
$ cp /lib/x86_64-linux-gnu/libc.so.6 /chroot/lib/
$ cp /lib64/ld-linux-x86-64.so.2 /chroot/lib64/
...
```

Выполнить вход в окружение:

```
$ chroot /chroot my-program
```

В результате процесс `my-program` не будет иметь доступ к большей части файловой системы ОС.

Примеры использования такого подхода в реальных системах:

- изоляция сетевых служб (FTP, DNS, SMTP и др.);
- recovery-среды, debug-билды, системные утилиты в ОС Linux (в частности, в Android OS).

1.4.2. Пример 2: Восстановление системы

`chroot` часто используется для восстановления системы, а также для создания загрузочных образов (см. [3]).

Типичный сценарий восстановления повреждённой системы:

1. Грузимся с помощью LiveCD.
2. Монтируем файловую систему повреждённой ОС, а также ряд служебных псевдофайловых систем:

   ```
   $ sudo mount /dev/sda2 /mnt
   $ sudo mount --bind /dev /mnt/dev
   $ sudo mount --bind /proc /mnt/proc
   $ sudo mount --bind /sys /mnt/sys
   ```

3. Запускаем `chroot`:

```
$ sudo chroot /mnt /bin/bash
...
```

4. Выполняем необходимые команды: например, установка пакетов, редактирование конфигурационных файлов, пересборка initramfs и т.п.

Подобный подход применялся нами для сброса пароля `grub`.

1.4.3. Пример 3: Тестирование библиотек

Аналогично тестированию приложения:

1. создается альтернативное `/lib`;
2. приложение запускается в `chroot`;
3. выполняется тестирование с альтернативным набором библиотек.

1.5. Заключение

- `chroot` - исторически важный механизм.
- Механизм `chroot` прост, но его возможности ограничены.
- Подходит для:
  - восстановления системы;
  - тестирования;
  - legacy-сценариев.
- Не должен рассматриваться как полноценная защита.
- Современные контейнеры являются логическим развитием идей `chroot`.

2. Механизм пространства имён Linux namespaces

2.1. Краткий исторический обзор

Пространства имён Linux (**Linux namespaces**) - механизм ядра, обеспечивающий изоляцию глобальных системных ресурсов между группами процессов [4][5][6].

- **2002–2003**: первым было реализовано **пространство файловой системы** (**mount namespace**, Linux 2.4.19), позволившее процессам иметь разные представления таблицы монтирования.
- **2006–2008**: были добавлены UTS, IPC, PID и Network namespaces.
- **2013 (Linux 3.8)**: завершена реализация поддержки пространств имён пользователей

(**user namespaces**), что позволило непривилегированным пользователям создавать пространства имён и создавать пользователей `root` внутри контейнера.

Пространства имён - фундамент контейнерных технологий (LXC, Docker, Podman), обеспечивающий лёгкую виртуализацию без запуска отдельного ядра.

2.2. Назначение механизма пространств имён

Основная идея — **изоляция глобальных ресурсов ядра**. Вместо одного общего пространства процессов, сети или единой файловой системы каждая группа процессов получает собственное представление этих ресурсов.

Namespaces применяются для:

- разрешения конфликты ресурсов (PID, порты, имена хостов);
- повышения безопасности за счёт изоляции процессов;
- запуска приложений, ожидающих «полноценную» систему - зачастую настроенную иначе, чем основная ("хостовая");
- лёгковесная виртуализация (вместо тяжёлых виртуальных машин, загружающих полноценную ОС).

2.3. Основные типы пространств имён

Пространство имён	**Что изолируется**
PID	Идентификаторы процессов (PID)
Network	Сетевой стек (интерфейсы, порты и проч.)
User	Идентификаторы пользователей (UID) и групп (GID)
Mount	Точки монтирования (части файловой системы)
IPC	SystemV IPC, очереди сообщений POSIX
UTS	Имя хоста (`hostname`) и доменное имя NIS (`domainname`)

2.4. Пространство имён Mount (CLONE_NEWNS)

Mount namespace реализует изолированное представление таблицы монтирования для процессов. Все операции `mount` и `umount` затрагивают только текущее пространство имён.

Пространство имён Mount необходим для:

- создания собственной корневой файловой системы (*root filesystem*) для контейнеров;
- изоляции рабочих директорий процессов;
- изоляции `/proc`, `/sys`, `tmpfs`.

С помощью же пространств имён Mount можно создавать полностью независимые файловые системы, ассоциируемые с различными процессами.

Для создания Mount namespace используется системный вызов clone(2) с флагом `CLONE_NEWNS`:

```
int clone(child_fn, child_stack, CLONE_NEWPID | CLONE_NEWNET | CLONE_NEWNS | SIGCHLD,
NULL);
```

При создании нового процесса с помощью clone(2) дочерний процесс переносится в новое пространство имён Mount, после чего любый операции монтирования не будут затрагивать ни родительский процесс, ни другие пространства имён.

Отличие от chroot

chroot(2) меняет только корневой каталог процесса, в отличие от пространства имён Mount, которое также изолирует таблицу монтирования, события файловой системы и проч.

Mount namespace - обобщение и расширение механизма `chroot`.

Пример: создание Mount namespace

```
$ sudo unshare --mount /bin/bash  ①
# export PS1="(mount ns) $PS1"
(mount ns) # mount --make-rprivate /  ②

(mount ns) # mount -t tmpfs tmpfs /mnt  ③

(mount ns) # mkdir /mnt/bin  ④
(mount ns) # cp /bin/busybox /mnt/bin/
(mount ns) # ln -s busybox /mnt/bin/sh
(mount ns) # ln -s busybox /mnt/bin/umount
(mount ns) # ln -s busybox /mnt/bin/touch
(mount ns) # ln -s busybox /mnt/bin/mkdir
(mount ns) # ln -s busybox /mnt/bin/rmdir
(mount ns) # ln -s busybox /mnt/bin/ls

(mount ns) # cd /mnt
(mount ns) # mkdir oldroot
(mount ns) # pivot_root . oldroot  ⑤

(mount ns) # umount -l /oldroot  ⑥
(mount ns) # rmdir /oldroot
(mount ns) # ls /
bin

(mount ns) # touch /from-newns.txt  ⑦
(mount ns) # /bin/mkdir -p /tmp/from-newns
(mount ns) # ls /
bin             from-newns.txt  tmp
(mount ns) # ls /tmp
from-newns
(mount ns) # exit
$ ls / | grep from-newns
```

```
$ ls /tmp | grep from-newns
```

① Создаём новый Mount namespace

② Изменяем тип namespace на private (обязательно для изоляции корневой FS!)

③ Подготавливаем новую корневую файловую систему (в данном случае `tmpfs` - временная FS, располагающаяся в RAM)

④ Создаём ссылки на универсальную утилиту `busybox` под именами необходимых далее утилит во вновь созданной FS

⑤ Переключаем корень `/` на подготовленную FS с помощью утилиты `pivot_root`

⑥ Удаляем старый корень FS

⑦ Проверяем корневую FS - файл и директория отсутствуют

2.5. Пространство имён PID (CLONE_NEWPID)

PID namespace изолирует пространство идентификаторов процессов. Каждый namespace имеет собственный PID 1.

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

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

В процессе загрузки ОС Linux:

- сначала создаётся процесс `init`, запускающий другие процессы и службы;
- в дереве процессов он является корневым, его идентификационный номер (PID) равен 1.

Мы наблюдали (и даже модифицировали) эту процедуру запуска на примере сброса пароля, когда перезаписывали `init=/bin/bash`.

Механизм namespaces позволяет создавать отдельное ответвление дерева процессов с собственным PID 1. Процесс, который создаёт такое ответвление, являются частью основного дерева (namespace), 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:

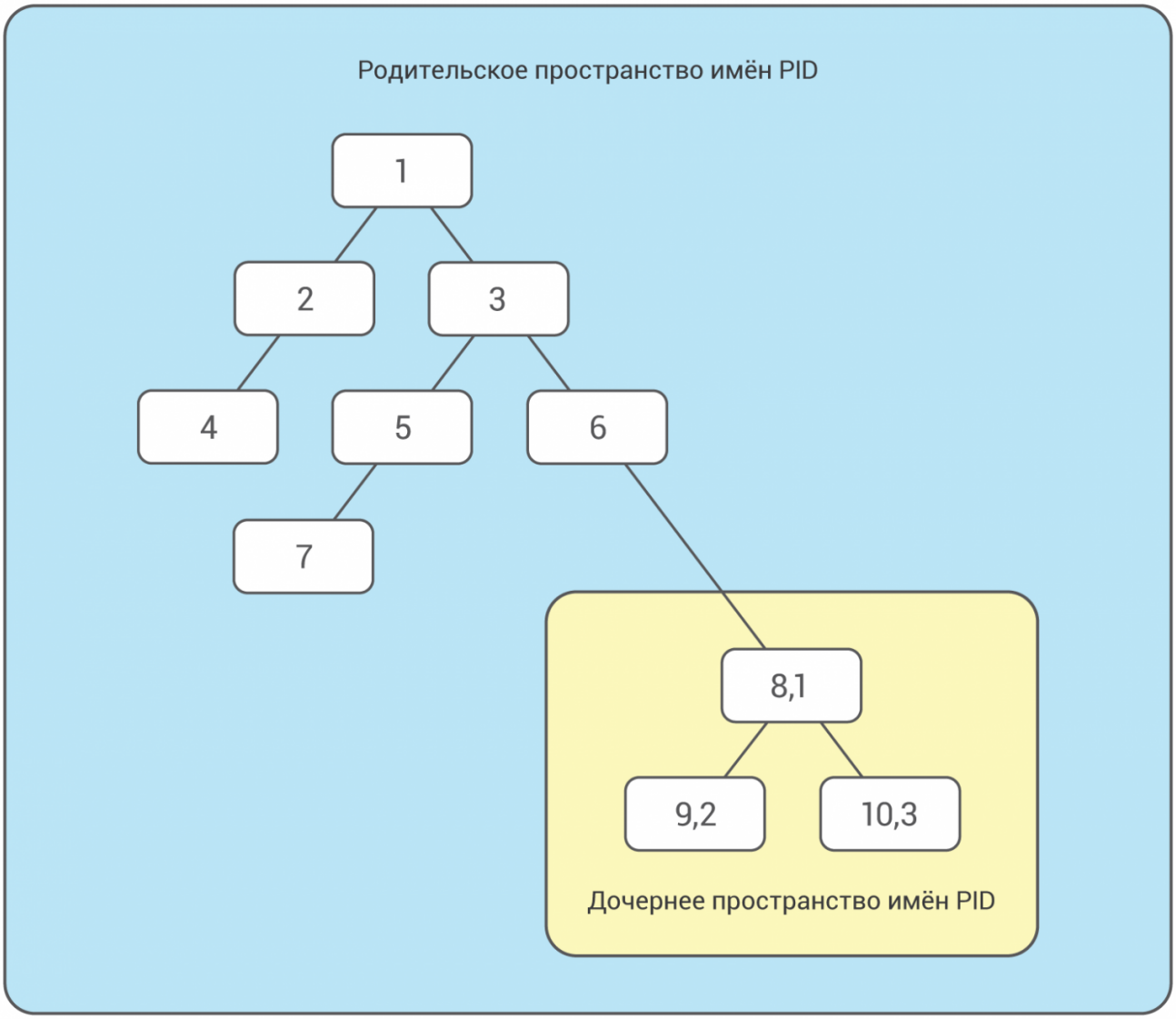

Figure 1. Пространство имён PID [6]

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 в каждом пространстве имён - **см. пары идентификаторов на схеме**).

Пространство имён PID может быть создано только в момент порождения нового процесса с помощью clone(2) с флагом `CLONE_NEWPID`. Как только clone(2) вызывается с этим флагом, новый процесс немедленно оказывается в новом пространстве имён PID, в рамках нового дерева процессов

При уничтожении PID namespace процесс с PID 1 должен собрать все zombie-процессы.

Пример: создание PID namespace в коде

Рассмотрим пример создания PID namespace в коде (файл `01_isolation__pid_namespace.c`).

```
1 #define _GNU_SOURCE
2 #include <sched.h>
```

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

static char child_stack[1048576];

static int child_fn() {
    printf("[child]  Parent's PID in the new PID namespace: %ld\n", (long)
getppid());
    printf("[child]  Child's PID in the new PID namespace:  %ld\n", (long)
getpid());
    return 0;
}

int main() {
    printf("[parent] Parent's PID in the main PID namespace: %ld\n", (long)
getpid());

    pid_t child_pid = clone(child_fn, child_stack+1048576, CLONE_NEWPID | SIGCHLD,
NULL);
    printf("[parent] Child's PID in the main PID namespace:  %ld\n", (long
)child_pid);

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

Скомпилируем и запустим:

```
$ cd examples
$ gcc -o pid_namespace 01_isolation__pid_namespace.c
$ ./pid_namespace
[parent] Parent's PID in the main PID namespace: 4295
[parent] Child's PID in the main PID namespace:  -1
$ sudo ./pid_namespace
[parent] Parent's PID in the main PID namespace: 4298
[parent] Child's PID in the main PID namespace:  4299
[child]  Parent's PID in the new PID namespace: 0
[child]  Child's PID in the new PID namespace:  1
```

Функция clone(2) создала новый процесс, клонировав текущий, и начала его выполнение. При этом для нового процесса было создано отдельное дерево процессов.

Обратите внимание:

- в исходном (основном) пространстве имён PID namespace идентификаторы "обычные" - потомки `init`;
- в новом PID namespace PID дочернего == 1, т.е. он как бы подменяет `init`, причём в новом

namespace у него нет родителя (PPID = 0).

Внесём в программу одно изменение и уберём флаг `CLONE_NEWPID` из вызова clone(2):

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

Перекомпилируем и запустим:

```
$ gcc -o pid_namespace 01_isolation__pid_namespace.c
$ ./pid_namespace
[parent] Parent's PID in the main PID namespace: 4371
[child]  Parent's PID in the new PID namespace: 4371
[child]  Child's PID in the new PID namespace:  4372
[parent] Child's PID in the main PID namespace:  4372
$ sudo ./pid_namespace
[parent] Parent's PID in the main PID namespace: 4375
[parent] Child's PID in the main PID namespace:  4376
[child]  Parent's PID in the new PID namespace: 4375
[child]  Child's PID in the new PID namespace:  4376
```

Теперь clone(2) просто создаёт новый процесс - **почти** также, как fork(2):

- fork(2) создаёт дочерний процесс, который представляет копию родительского, однако адресные пространства независимы;
- clone(2) не просто создаёт копию, но позволяет разделять элементы контекста выполнения между дочерним и родительским процессами;

В приведённом выше примере с clone(2) используется аргумент `child_stack`, который задаёт положение стека для дочернего процесса. Так как дочерний и родительский процессы могут разделять адресное пространство, дочерний процесс не может выполняться на том же стеке, что и родительский. Поэтому родительский процесс должен установить пространство памяти для стека дочернего процесса и передать указатель на него в вызове clone(2).

Пример: создание вложенных PID namespace

Рассмотрим пример создания вложенных PID namespace в коде (файл `02_isolation__nested_pid_namespace.c`).

```
1 #define _GNU_SOURCE
2 #include <sched.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <sys/wait.h>
6 #include <unistd.h>
7
8 static int delay = 60;
9 static char child_stack[1048576];
```

```c
static char child_stack2[1048576];


static int child_fn2() {
    printf("[child 2] Child's PID in the second new PID namespace:      %ld\n", (long)getppid());
    printf("[child 2] Grandchild's PID in the second new PID namespace: %ld\n", (long)getpid());

    printf("[child 2] Sleeping for %d seconds...\n", delay);
    sleep(delay);
    printf("[child 2] Done\n");

    return 0;
}

static int child_fn() {
    printf("[child 1] Parent's PID in the new PID namespace:            %ld\n", (long)getppid());
    printf("[child 1] Child's PID in the new PID namespace:             %ld\n", (long)getpid());

    printf("[child 1] Spawning a grandchild process...\n");

    pid_t child_pid = clone(child_fn2, child_stack2+1048576, CLONE_NEWPID | SIGCHLD, NULL);
    printf("[child 1] Grandchild's PID in the new PID namespace:        %ld\n", (long)child_pid);

    waitpid(child_pid, NULL, 0);
    printf("[child 1] Done\n");

    return 0;
}

int main(int argc, const char* argv[]) {
    if (argc > 1) {
        delay = atoi(argv[1]);
    }

    printf("[parent]  Parent's PID in the main PID namespace:           %ld\n", (long)getpid());

    pid_t child_pid = clone(child_fn, child_stack+1048576, CLONE_NEWPID | SIGCHLD, NULL);
    printf("[parent]  Child's PID in the main PID namespace:            %ld\n", (long)child_pid);

    waitpid(child_pid, NULL, 0);
    printf("[parent]  Done\n");

```

```
52     return 0;
53 }
```

Скомпилируем и запустим:

```
$ gcc -o nested_pid_namespace 02_isolation__nested_pid_namespace.c

$ ./nested_pid_namespace 3
[parent]  Parent's PID in the main PID namespace:           4554
[parent]  Child's PID in the main PID namespace:            -1  ①
[parent]  Done

$ sudo ./nested_pid_namespace 180 &
[parent]  Parent's PID in the main PID namespace:           4633  ②
[parent]  Child's PID in the main PID namespace:            4634  ③
[child 1] Parent's PID in the new PID namespace:            0
[child 1] Child's PID in the new PID namespace:             1     ③
[child 1] Spawning a grandchild process...
[child 1] Grandchild's PID in the new PID namespace:        2     ④
[child 2] Child's PID in the second new PID namespace:      0
[child 2] Grandchild's PID in the second new PID namespace: 1     ④
[child 2] Sleeping for 180 seconds...

$ sudo ps -o pid,pidns,args -p 4633  ⑤
    PID      PIDNS COMMAND
   4633 4026531836 ./nested_pid_namespace 180
$ sudo ps -o pid,pidns,args -p 4634
    PID      PIDNS COMMAND
   4634 4026532386 ./nested_pid_namespace 180
$ sudo ps -o pid,pidns,args -p 4635
    PID      PIDNS COMMAND
   4635 4026532387 ./nested_pid_namespace 180

$ pstree -a -p 4643  ⑥
nested_pid_name,4633 180
  └─nested_pid_name,4634 180
      └─nested_pid_name,4635 180
$ sudo ps --forest -a -o pid,pidns,args  ⑥
    PID      PIDNS COMMAND
   4643 4026531836 ps --forest -a -o pid,pidns,args
   4633 4026531836 ./nested_pid_namespace 180
   4634 4026532386  \_ ./nested_pid_namespace 180
   4635 4026532387      \_ ./nested_pid_namespace 180
...

$ sudo ls -lai /proc/4633/ns | grep pid  ⑦
28036 lrwxrwxrwx 1 root root 0 Dec 23 03:20 pid -> 'pid:[4026531836]'
28255 lrwxrwxrwx 1 root root 0 Dec 23 03:22 pid_for_children -> 'pid:[4026531836]'
$ sudo ls -lai /proc/4634/ns  | grep pid
28049 lrwxrwxrwx 1 root root 0 Dec 23 03:20 pid -> 'pid:[4026532386]'
```

```
32025 lrwxrwxrwx 1 root root 0 Dec 23 03:22 pid_for_children -> 'pid:[4026532386]'
$ sudo ls -lai /proc/4635/ns | grep pid
28062 lrwxrwxrwx 1 root root 0 Dec 23 03:20 pid -> 'pid:[4026532387]'
32037 lrwxrwxrwx 1 root root 0 Dec 23 03:22 pid_for_children -> 'pid:[4026532387]'

$ sudo lsns -l | grep pid  ⑧
4026531836 pid       197     1 root            /sbin/init splash
4026532386 pid         1  4634 root            ./nested_pid_namespace 180
4026532387 pid         1  4635 root            ./nested_pid_namespace 180

$ sudo ls -lai /proc/1/ns | grep pid
25901 lrwxrwxrwx 1 root root 0 Dec 23 03:17 pid -> pid:[4026531836]
32926 lrwxrwxrwx 1 root root 0 Dec 23 03:35 pid_for_children -> pid:[4026531836]
```

① clone(2) без привилегий не выполнился

② Запускаем с правами `root`

③ Дочернему процессу назначен PID 4634 в основном namespace и PID 1 в новом (первом созданном)

④ Процессу-внуку назначен PID 2 в первом созданном PID NS и PID 1 в новом (втором созданном) PID NS

⑤ Все три процесса иерархии живут в своих PID NS

⑥ Утилиты `pstree` и `ps` с флагом `--forest` позволяют увидеть PID NS процессов

⑦ PID NS также можно найти с помощью `procfs`

⑧ Специальная утилита `lsns` позволяет просматривать текущие namespaces в системе

2.6. Сетевое пространство имён (CLONE_NEWNET)

Network namespace предоставляет отдельный сетевой стек:

- интерфейсы;
- IP-адреса;
- маршруты;
- firewall;
- пространство портов.

Благодаря пространству имён NET мы можем выделять для изолированных процессов собственные сетевые интерфейсы, даже loopback-интерфейс для каждого пространства имён будет отдельным.

После создания Net namespace содержит только интерфейс loopback, причём в выключенном состоянии.

Сетевые пространства имён можно создавать с помощью:

- системного вызова clone(2) с флагом `CLONE_NEWNET`;

- системного вызова unshare(2): `unshare` позволяет процессу или потоку отделять части контекста исполнения, общие с другими процессами (потоками).

Помещения процессов в новое сетевое пространство имён возможно следующими способами:

- Процесс, создавший новое пространство имён, может порождать другие процессы, и каждый из этих процессов будет наследовать сетевое пространство имён родителя;
- В ядре имеется специальный системный вызов — setns(2), с его помощью можно поместить вызывающий процесс или поток 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(2).
- При создании нового пространства имён с помощью команды ip создаётся файл в директории `/var/run/netns/`. Чтобы получить файловый дескриптор, достаточно просто открыть этот файл.

Пример: создание Net namespace

```
$ sudo ip netns add ns1  ①
$ sudo ip netns exec ns1 ip addr  ②
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default qlen 1000  ③
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
$ sudo ip netns exec ns1 ping 127.0.0.1  ④
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
^C
--- 127.0.0.1 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms

$ sudo ip netns exec ns1 ip link set lo up  ⑤
$ sudo ip netns exec ns1 ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen
1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever

$ sudo ip netns exec ns1 ping 127.0.0.1  ⑥
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.029 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.019 ms
^C
--- 127.0.0.1 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1004ms
rtt min/avg/max/mdev = 0.019/0.024/0.029/0.005 ms

$ sudo ip netns delete ns1  ⑦
```

① Создаём сетевое пространство имён

② Выводим список интерфейсов нового Net NS

③ В новом Net NS создан всего один интерфейс - loopback, причём изначально он выключен

④ Интерфейс loopback не пингуется

⑤ Включаем loopback

⑥ Теперь пинги пошли

⑦ Удаляем сетевое пространство имён

Пример: соединение с хостом через veth

Настраиваем пару интерфейсов `veth`, создаём Net namespace `ns2` и помещаем один интерфейс в `ns2`, настраиваем адреса:

```
$ sudo ip netns add ns2  ①
$ sudo ip link add veth0 type veth peer name veth1  ②
$ ip link  ③
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: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP mode
DEFAULT group default qlen 1000
    link/ether 08:00:27:89:d9:83 brd ff:ff:ff:ff:ff:ff
5: veth1@veth0: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noop state DOWN mode
DEFAULT group default qlen 1000
    link/ether b2:6b:0e:4c:93:6e brd ff:ff:ff:ff:ff:ff
6: veth0@veth1: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noop state DOWN mode
DEFAULT group default qlen 1000
    link/ether fa:6d:62:a6:44:9f brd ff:ff:ff:ff:ff:ff

$ sudo ip link set veth1 netns ns2  ④
$ ip link  ⑤
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: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP mode
DEFAULT group default qlen 1000
    link/ether 08:00:27:89:d9:83 brd ff:ff:ff:ff:ff:ff
6: veth0@if5: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group
default qlen 1000
    link/ether fa:6d:62:a6:44:9f brd ff:ff:ff:ff:ff:ff link-netns ns2

$ sudo ip addr add 10.0.0.1/24 dev veth0  ⑥
$ sudo ip link set veth0 up
$ ip link show veth0
6: veth0@if5: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP mode
DEFAULT group default qlen 1000
    link/ether fa:6d:62:a6:44:9f brd ff:ff:ff:ff:ff:ff link-netns ns2
```

```
$ sudo ip netns exec ns2 ip addr add 10.0.0.2/24 dev veth1  ⑦
$ sudo ip netns exec ns2 ip link set veth1 up
$ sudo ip netns exec ns2 ip link
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
5: veth1@if6: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP mode
DEFAULT group default qlen 1000
    link/ether b2:6b:0e:4c:93:6e brd ff:ff:ff:ff:ff:ff link-netnsid 0

$ sudo ip netns exec ns2 ip route add default via 10.0.0.2 dev veth1  ⑧
$ sudo ip netns exec ns2 ip route
default via 10.0.0.2 dev veth1
10.0.0.0/24 dev veth1 proto kernel scope link src 10.0.0.2
```

① Создаём пространство имён `ns2`

② Создаём пару виртуальных интерфейсов `veth`

③ Видим в списке созданных интерфейсов пару `veth` (#5 и #6)

④ Добавляем `venth1` в Net namespace `ns2`

⑤ Видим, что из списка интерфейсов основного сетевого пространства `veth1` пропал

⑥ Конфигурируем адрес 10.0.0.1/24 на интерфейсе `veth0`

⑦ Конфигурируем адрес 10.0.0.2/24 на интерфейсе `veth1`

⑧ Настраиваем default gateway для `veth1`

Яростно пингуем из сетевого интерфейса `ns2` мост (`veth0`) и хост, располагающиеся в основном сетевом пространстве:

```
$ sudo ip netns exec ns2 ping 10.0.0.1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
64 bytes from 10.0.0.1: icmp_seq=1 ttl=64 time=0.035 ms
^C
--- 10.0.0.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.035/0.035/0.035/0.000 ms
$ sudo ip netns exec ns2 ping 10.0.2.15

PING 10.0.2.15 (10.0.2.15) 56(84) bytes of data.
64 bytes from 10.0.2.15: icmp_seq=1 ttl=64 time=0.047 ms
64 bytes from 10.0.2.15: icmp_seq=2 ttl=64 time=0.023 ms
^C
--- 10.0.2.15 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1027ms
rtt min/avg/max/mdev = 0.023/0.035/0.047/0.012 ms
```

2.7. Пространство имён UTS (CLONE_NEWUTS)

UTS namespace изолирует системные идентификаторы: имя хоста (`hostname`) и доменное имя NIS (`domainname`).

Пример

```
$ sudo unshare --uts /bin/bash  ①
$ sudo hostname container1  ②
$ sudo hostname  ③
sudo: unable to resolve host container: Temporary failure in name resolution
container
# exit
$ sudo hostname  ④
OSSec-L2
```

① Создаём UTS namespace

② Изменяем имя хоста

③ Имя хоста изменилось в созданном UTC namespace

④ Однако имя хоста остаётся неизменным в основном пространстве имён

2.8. Пространство имён пользователей (User Namespace)

Виртуализация привилегий

User namespace изолирует **идентификаторы пользователей и групп** (UID/GID). Он позволяет процессу иметь UID 0 (root) **внутри контейнера**, оставаясь непривилегированным пользователем на хосте.

Отображение UID/GID (UID/GID mapping)

В User namespace используется явное отображение идентификаторов между контейнером и хостом:

```
Container UID 0  --->  Host UID 100000
Container UID 1  --->  Host UID 100001
```

Настройка выполняется через `procfs`:

- `/proc/[pid]/uid_map`;
- `/proc/[pid]/gid_map`.

Пример: User namespace без root на хосте

```
$ sudo unshare --user --map-root-user /bin/bash
# id
```

```
uid=0(root) gid=0(root) groups=0(root)
```

В результате UID пользователя:

- внутри User namespace: `uid=0(root)`;
- на хосте: обычный пользователь.

Ограничения механизма User namespace

- Не все файловые системы поддерживают user namespaces.
- Неправильная настройка может привести к отказу сервисов.
- Исторически User namespaces были источником багов в ядре.

2.9. Linux namespaces в задачах обеспечения безопасности

Что дают пространства имён с точки зрения безопасности

- Ограничение видимости ресурсов.
- Снижение последствий компрометации процесса.
- Изоляция сетевых атак и sniffing.

Важные ограничения

- Все пространства имён **делят одно ядро**.
- Уязвимость в ядре = риск для всех контейнеров.
- Namespace ≠ sandbox по умолчанию.

User namespace и привилегии

- root внутри user namespace ≠ root на хосте.
- Существенно снижает риск при запуске контейнеров.
- Требует осторожной настройки (особенно на серверах).

Best practices

- Всегда комбинировать namespaces с cgroups.
- Минимизировать capabilities.
- Использовать seccomp и LSM (SELinux, AppArmor).

2.10. Домашний контейнер - пример совместного использования cgroup, seccomp и Linux namespaces

```
(CPU limit + pinning)
```

1. Создаём окружение

```
$ sudo mkdir -p /container/root
$ sudo chown $USER:$USER -R /container

$ cp examples/04_container.c /container/

$ wget https://dl-cdn.alpinelinux.org/alpine/v3.23/releases/x86_64/alpine-minirootfs-
3.23.2-x86_64.tar.gz
$ tar xvf alpine-minirootfs-3.23.2-x86_64.tar.gz -C /container/root

$ tree /container/ -L 2
/container/
├── 04_container.c
└── root
    ├── bin
    ├── dev
    ├── etc
    ├── home
    ├── lib
    ├── media
    ├── mnt
    ├── opt
    ├── proc
    ├── root
    ├── run
    ├── sbin
    ├── srv
    ├── sys
    ├── tmp
    ├── usr
    └── var
```

2. Компилируем `container` и выдаём необходимые Capabilities

```
$ gcc -O2 -Wall 04_container.c -o container -lcap
```

3. Создаём `cgroup` `mycontainer` и настраиваем лимиты

```
$ sudo mkdir /sys/fs/cgroup/mycontainer
$ echo "+cpu +cpuset" | sudo tee /sys/fs/cgroup/cgroup.subtree_control

$ # 70% CPU (например, 70ms из 100ms)
$ echo "70000 100000" | sudo tee /sys/fs/cgroup/mycontainer/cpu.max

$ # прикрепляем процессы на CPU #1
$ echo 1 | sudo tee /sys/fs/cgroup/mycontainer/cpuset.cpus
$ echo 0 | sudo tee /sys/fs/cgroup/mycontainer/cpuset.mems
```

4. Создание и настройка Network namespace и `veth`

```
$ # создаём пару veth, конфигурируем
$ sudo ip link add veth-host type veth peer name veth-cont

$ sudo ip addr add 172.16.42.1/24 dev veth-host
$ sudo ip link set veth-host up

$ sudo ip addr add 172.16.42.2/24 dev veth-cont
$ sudo ip link set veth-host up

$ # настраиваем source NAT
$ sudo sysctl -w net.ipv4.ip_forward=1
$ sudo iptables -t nat -A POSTROUTING -s 172.16.42.0/24 -o enp0s3 -j MASQUERADE
```

5. Запускаем контейнер

```
$ sudo ./container

/ # ip addr add 172.16.42.2/24 dev veth-cont
/ # ip link set veth-cont up
/ # ip a
/ # ip route add default via 172.16.42.1
/ # ping 172.16.42.1
/ # ping 10.0.2.15
/ # ping 1.1.1.1

/ # echo "Hello, container!" > /home/
```

Источники

- [1] [Wikipedia] chroot
- [2] [Arch Wiki] chroot
- [3] [github.com] How to create a custom Ubuntu live from scratch
- [4] [lwn.net] Namespaces in operation, part 1: namespaces overview
- [5] [toptal.com] Separation Anxiety: A Tutorial for Isolating Your System with Linux Namespaces
- [6] [habr.ru] Механизмы контейнеризации: namespaces
- [7] [habr.ru] Ifeanyi Ubah. Глубокое погружение в Linux namespaces (Перевод) Часть 1 Часть 2 Часть 3 Часть 4
- [8] [Red Hat Blog] Steve Ovens. Building a container by hand using namespaces Часть 1 Часть 2 Часть 3 Часть 4 Часть 5 Часть 6