

Министерство науки и высшего образования Российской Федерации  
Федеральное государственное автономное образовательное учреждение  
высшего образования  
«Санкт-Петербургский политехнический университет Петра Великого»

—  
Институт компьютерных наук и кибербезопасности  
**Высшая школа кибербезопасности**

## **КУРСОВАЯ РАБОТА**

Изоляция ресурсов с помощью пространств имен: создание мини-  
контейнера  
по дисциплине «Операционные системы»

Выполнил  
студент гр. 5131001/30002

Мишенёв Н. С.

Руководитель  
программист

Огнёв Р. А.

«\_\_\_» \_\_\_\_\_ 2025 г.

Санкт-Петербург  
2025г.

## СОДЕРЖАНИЕ

ВВЕДЕНИЕ.....	3
ОСНОВНАЯ ЧАСТЬ.....	4
ТЕОРЕТИЧЕСКАЯ ЧАСТЬ.....	4
1. Пространство имён или namespaces.....	4
2. Типы пространств имен в Linux.....	5
3. Создание пространств имён.....	11
3.1. Unshare.....	11
3.2. clone(...). .....	12
4. Примеры использования пространств имён.....	13
5. Процесс создания нового пространства имён.....	13
ПРАКТИЧЕСКАЯ ЧАСТЬ.....	17
Разработка программы контейнеризатора на C++.....	17
1. UTS namespace.....	18
2. USER namespace.....	18
3. MNT namespace.....	20
4. PID namespace.....	22
ЗАКЛЮЧЕНИЕ.....	25
ПРИЛОЖЕНИЕ А. ИСХОДНЫЙ КОД РАЗРАБОТАННОГО ПРИЛОЖЕНИЯ.....	26

## ВВЕДЕНИЕ

### Цель

Изучение механизма пространств имен (namespaces) в Linux, лежащего в основе контейнеризации.

### Задачи

- Изучить типы пространств имен в Linux (PID, Network, Mount, UTS, IPC, User).
- Рассмотреть создание namespaces через *clone* (с флагами `CLONE_NEW*`) и *unshare*.
- Привести примеры использования пространства имён.
- Написать программу на C, которая запускает процесс в отдельных *namespaces*.
- Сделать выводы о проделанной работе

# ОСНОВНАЯ ЧАСТЬ

## ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

### 1. Пространство имён или namespaces.

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

Другими словами, пространства имен определяют набор ресурсов, которые может использовать процесс, поэтому намеренно скрыв от процесса какие-либо ресурсы, поместив их в другой namespace, мы можем запретить процессу ими пользоваться.

На высоком уровне они позволяют тонко разделять глобальные ресурсы операционной системы, такие как точки монтирования, сетевой стек и утилиты межпроцессного взаимодействия. В Linux они обычно представлены как файлы в директории `/proc/<pid>/ns`. (Рисунок 1)

```
clowixdev@clowixdev ~> ps
  PID TTY          TIME CMD
   852 pts/0    00:00:00 fish
  1166 pts/0    00:00:00 ps
clowixdev@clowixdev ~> ls /proc/852/ns -al
total 0
dr-x--x--x 2 clowixdev clowixdev 0 May 26 11:25 .
dr-xr-xr-x 9 clowixdev clowixdev 0 May 26 11:24 ..
lrwxrwxrwx 1 clowixdev clowixdev 0 May 26 11:25 cgroup -> 'cgroup:[4026531835]'
lrwxrwxrwx 1 clowixdev clowixdev 0 May 26 11:25 ipc -> 'ipc:[4026532242]'
lrwxrwxrwx 1 clowixdev clowixdev 0 May 26 11:25 mnt -> 'mnt:[4026532253]'
lrwxrwxrwx 1 clowixdev clowixdev 0 May 26 11:25 net -> 'net:[4026531840]'
lrwxrwxrwx 1 clowixdev clowixdev 0 May 26 11:25 pid -> 'pid:[4026532255]'
lrwxrwxrwx 1 clowixdev clowixdev 0 May 26 11:25 pid_for_children -> 'pid:[4026532255]'
lrwxrwxrwx 1 clowixdev clowixdev 0 May 26 11:25 time -> 'time:[4026531834]'
lrwxrwxrwx 1 clowixdev clowixdev 0 May 26 11:25 time_for_children -> 'time:[4026531834]'
lrwxrwxrwx 1 clowixdev clowixdev 0 May 26 11:25 user -> 'user:[4026531837]'
lrwxrwxrwx 1 clowixdev clowixdev 0 May 26 11:25 uts -> 'uts:[4026532254]'
clowixdev@clowixdev ~>
```

Рисунок 1 – пространство имён для запущенного терминала

Сильная сторона пространств имен в том, что они ограничивают доступ к системным ресурсам без информирования об этом выполняющегося процесса.

## 2. Типы пространств имен в Linux.

В Linux существует несколько типов пространств имён. При запуске системы, инициализируется по 1 экземпляру каждого типа пространства имён, кроме пространства имён файловой системы. После инициализации, можно объединять или создавать дополнительные пространства имён.

Все пространства имён поддерживают вложенность, то есть между ними можно установить связь «родитель — потомок». Таким образом некоторые пространства наследуют все свойства от своего родительского пространства имён. Однако это верно не для всех пространств.

Функциональные возможности пространства имён одинаковы для всех типов: каждый процесс связан с пространством имён и может видеть или использовать только ресурсы, связанные с этим пространством имён, и, где это применимо, — с его потомками.

Рассмотрим типы пространств имён в Linux-системе:

### 2.1. PID

Исторически ядро Linux поддерживает одно дерево процессов. Древовидная структура данных содержит ссылку на каждый активный процесс в виде иерархии от родителя к потомку. Она также нумерует все выполняющиеся в ОС процессы (Рисунок 2). Эта структура поддерживается в файловой системе **procfs**, которая является свойством исключительно работающей ОС.

```
clowixdev@clowixdev ~> ls /proc/
1      140    333    cmdline      ioports      misc          thread-self
111    145    334    config.gz    irq          modules       timer_list
114    166    366    consoles    kallsyms     mounts        tty
118    1921   67     cpuinfo      kcore        mtrr          uptime
1181   2      7      crypto       key-users    net           version
1183   218   723    devices      keys         pagetypeinfo vmallocinfo
119    2526   850    diskstats    kmsg         partitions    vmstat
1194   289   851    dma          kpagecgroup schedstat     zoneinfo
1197   308   852    driver       kpagecount   self
1198   327   93     execdomains  kpageflags   softirqs
129    328    acpi      filesystems  loadavg      stat
1297   329    buddyinfo fs           locks        swaps
1298   331    bus       interrupts   mdstat       sys
136    332    cgroups   iomem        meminfo      sysvipc
clowixdev@clowixdev ~>
```

Рисунок 2 – файловая система **procfs**

Эта структура позволяет процессам с достаточными привилегиями прикрепляться к другим процессам, инспектировать эти процессы, обмениваться с ними информацией и/или завершать их.

Она также содержит информацию о корневом каталоге процесса, его текущем рабочем каталоге, дескрипторах открытых файлов, адресах виртуальной памяти, доступных точках монтирования и т.д. Пример структуры **procfs** и дерева процессов представлен на рисунке 3 и 4 соответственно:

```
clowixdev@clowixdev ~> echo "Пример структуры procfs"
Пример структуры procfs
clowixdev@clowixdev ~> sudo ls /proc/1/
arch_status      environ          mem              personality      stat
attr             exe              mountinfo        projid_map       statm
auxv             fd               mounts           root             status
cgroup           fdinfo           mountstats       sched            syscall
clear_refs       gid_map          net              schedstat        task
cmdline          io               ns               sessionid        timers_offsets
comm             limits           oom_adj          setgroups        timers
coredump_filter  loginuid         oom_score        smaps            timerslack_ns
cpuset           map_files        oom_score_adj    smaps_rollup     uid_map
cwd              maps             pagemap          stack            wchan
clowixdev@clowixdev ~>
```

Рисунок 3 – пример структуры **procfs** процесса с **PID 1**

```

clowixdev@clowixdev ~> pstree | head -n 15
systemd--2*[agetty]
|
|-cron
|-dbus-daemon
|-dnsmasq---dnsmasq
|-init-systemd(Ub--SessionLeader---Relay(852)---fish--head
|                                     |-pstree
|                                     `--{fish}
|
|                                     |-init---{init}
|                                     `--{init-systemd(Ub}
|
|-libvirtd---18*[{libvirtd}]
|-networkd-dispat
|-packagekitd---2*[{packagekitd}]
|-polkitd---2*[{polkitd}]
|-rsyslogd---3*[{rsyslogd}]
|-snapd---16*[{snapd}]
clowixdev@clowixdev ~>

```

Рисунок 4 – пример структуры дерева процессов.

## 2.2. Network (NET)

Сетевое пространство имен ограничивает видимость процесса внутри сети. Оно позволяет процессу располагать собственной частью сетевого стека хоста (набором сетевых интерфейсов и правилами маршрутизации) (Рисунок 5)

```

clowixdev@clowixdev ~> ip link
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: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1280 qdisc mq state UP mode DEFAUL
   T group default qlen 1000
    link/ether 00:15:5d:af:26:6a brd ff:ff:ff:ff:ff:ff
3: virbr0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN
   mode DEFAULT group default qlen 1000
    link/ether 52:54:00:cb:6a:45 brd ff:ff:ff:ff:ff:ff
clowixdev@clowixdev ~> ip route
default via 172.20.144.1 dev eth0 proto kernel
172.20.144.0/20 dev eth0 proto kernel scope link src 172.20.154.132
192.168.122.0/24 dev virbr0 proto kernel scope link src 192.168.122.1 linkdown
clowixdev@clowixdev ~>

```

Рисунок 5 – пример сетевых интерфейсов и правил маршрутизации

## 2.3. Mount (MNT)

Пространства имен mount (MNT) позволяют создавать деревья файловых систем под отдельные процессы, тем самым создавая

представления корневой файловой системы. Linux поддерживает структуру данных для всех различных файловых систем, смонтированных в системе.

Эта структура является индивидуальной для каждого процесса, а также пространства имен. В нее входит информация о том, какие разделы дисков смонтированы, где они смонтированы и тип монтирования (**RO/RW**). (Рисунок 6)

```
clowixdev@clowixdev ~> cat /proc/1/mounts
none /usr/lib/modules/5.15.167.4-microsoft-standard-WSL2 overlay rw,nosuid,nodev,noatime,lowerdir=/modules,upperdir=/lib/modules/5.15.167.4-microsoft-standard-WSL2/rw/upper,workdir=/lib/modules/5.15.167.4-microsoft-standard-WSL2/rw/work 0 0
none /mnt/wsl tmpfs rw,relatime 0 0
drivers /usr/lib/wsl/drivers 9p ro,dirsync,nosuid,nodev,noatime,aname=drivers;fmask=222;dmask=222,mmap,access=client,msize=65536,trans=fd,rfd=8,wfd=8 0 0
/dev/sdc / ext4 rw,relatime,discard,errors=remount-ro,data=ordered 0 0
none /mnt/wslg tmpfs rw,relatime 0 0
/dev/sdc /mnt/wslg/distro ext4 ro,relatime,discard,errors=remount-ro,data=ordered 0 0
none /usr/lib/wsl/lib overlay rw,nosuid,nodev,noatime,lowerdir=/gpu_lib_packaged:/gpu_lib_inbox,upperdir=/gpu_lib/rw/upper,workdir=/gpu_lib/rw/work 0 0
rootfs /init rootfs ro,size=8149172k,nr_inodes=2037293 0 0
none /dev devtmpfs rw,nosuid,relatime,size=8149172k,nr_inodes=2037293,mode=755 0 0
sysfs /sys sysfs rw,nosuid,nodev,noexec,noatime 0 0
proc /proc proc rw,nosuid,nodev,noexec,noatime 0 0
```

Рисунок 6 – MNT структура для процесса с PID 1

Пространства имен в Linux дают возможность копировать эту структуру данных и передавать копию разным процессам. Таким образом, эти процессы могут изменять данную структуру (монтировать и размонтировать), не влияя на точки монтирования друг друга.

Предоставляя разные копии структуры файловой системы, ядро изолирует список точек монтирования, видимых процессу в пространстве имен. (Рисунок 7)



```
clowixdev@clowixdev ~> cat /proc/self/mountinfo | head -n 3
73 78 0:28 / /usr/lib/modules/5.15.167.4-microsoft-standard-WSL2 rw,nosuid,nodev,noatime - overlay none rw,lowerdir=/modules,upperdir=/lib/modules/5.15.167.4-microsoft-standard-WSL2/rw/upper,workdir=/lib/modules/5.15.167.4-microsoft-standard-WSL2/rw/work
74 78 0:31 / /mnt/wsl rw,relatime shared:1 - tmpfs none rw
75 78 0:33 / /usr/lib/wsl/drivers ro,nosuid,nodev,noatime - 9p drivers ro,directsync,aname=drivers;fmask=222;dmask=222,mmap,access=client,msize=65536,trans=fd,rfd=8,wfd=8
clowixdev@clowixdev ~> █
```

Рисунок 7 – список точек монтирования в namespace текущего процесса

## 2.4. UTS

Пространство имен UTS изолирует имя хоста системы для определенного процесса.

Большая часть взаимодействия с хостом выполняется через IP-адрес и номер порта. Однако для человеческого восприятия все сильно упрощается, когда у процесса имя. К примеру, выполнять поиск по файлам журналов гораздо проще, когда определено имя хоста. Также это связано с тем, что в динамической среде IP могут изменяться. (Рисунок 8, Рисунок 9)

```
clowixdev@clowixdev ~> hostname
clowixdev
clowixdev@clowixdev ~> sudo unshare -u /bin/fish
Welcome to fish, the friendly interactive shell
Type help for instructions on how to use fish
root@clowixdev /h/clowixdev# hostname test; hostname
test
root@clowixdev /h/clowixdev# █
```

Рисунок 8 – изменение **hostname** в другом UTS namespace

```
clowixdev@clowixdev ~> hostname
clowixdev
clowixdev@clowixdev ~> █
```

Рисунок 9 – проверка **hostname** в другом терминале

## 2.5. IPC

Пространство имен **IPC** предоставляет изоляцию для механизмов взаимодействия процессов, таких как семафоры, очереди сообщений, разделяемая память и т.д.

Обычно, когда процесс ответвляется, он наследует все **IPC**, открытые его родителем. Процессы внутри **IPC namespace** не могут видеть или взаимодействовать с ресурсами **IPC** вышестоящего пространства имен. Пример структуры **ipcs** приведён на рисунке 10:

```
clowixdev@clowixdev ~> ipcs
----- Message Queues -----
key          msqid      owner          perms         used-bytes   messages
----- Shared Memory Segments -----
key          shmid      owner          perms         bytes        nattch       status
----- Semaphore Arrays -----
key          semid      owner          perms         nsems

clowixdev@clowixdev ~> █
```

Рисунок 10 – пример IPC namespace для текущего процесса

## 2.6. USER

Все процессы в Linux имеют родительский процесс. Также существуют привилегированные и непривилегированные процессы, что определяется их пользовательским **ID** или же **UID**. В зависимости от этого **UID** процессы получают разные привилегии в ОС. Пользовательское пространство имен – это функционал ядра, позволяющий выполнять виртуализацию этого атрибута для каждого процесса.

Пользовательские пространства имен изолируют связанные с безопасностью идентификаторы и атрибуты. В частности, **ID** пользователей, групповые **ID**, корневой каталог, ключи и возможности.

Для примера, запустим процесс в новом **USER namespace** и посмотрим на его родителя (Рисунок 11)

```
clowixdev@clowixdev ~> id
uid=1000(clowixdev) gid=1000(clowixdev) groups=1000(clowixdev),4(adm),20(dialout),
24(cdrom),25(floppy),27(sudo),29(audio),30(dip),44(video),46(plugdev),118(netdev),
121(libvirt)
clowixdev@clowixdev ~> ps -l
F S  UID      PID     PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S  1000      852      851  0  80   0 - 40329 futex_ pts/0        00:00:01 fish
0 R  1000     3054      852  0  80   0 - 2617 -      pts/0        00:00:00 ps
clowixdev@clowixdev ~> unshare -U /bin/fish
Welcome to fish, the friendly interactive shell
Type help for instructions on how to use fish
clowixdev@clowixdev ~> id
uid=65534(nobody) gid=65534(nogroup) groups=65534(nogroup)
clowixdev@clowixdev ~> ps -l
F S  UID      PID     PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S  65534     852      851  0  80   0 - 40333 -      pts/0        00:00:01 fish
0 S  65534    3070      852  0  80   0 - 40160 futex_ pts/0        00:00:00 fish
0 R  65534    3106    3070  0  80   0 - 2617 -      pts/0        00:00:00 ps
clowixdev@clowixdev ~> █
```

Рисунок 11 – пример создания нового пользовательского пространства имён

Как можно заметить, **PID** родительского процесса не изменился, но его обладатель теперь кто-то, о ком нет информации, потому что у запущенного процесса в новом **namespace**, нет доступа за его пределы.

### 3. Создание пространств имён.

Создание пространств имён происходит с помощью системного вызова **clone()** и с помощью команды **unshare**. Рассмотрим самую суть способов создания пространств и их отличия:

#### 3.1. Unshare

Команда **unshare** запускает выбранный процесс в новом **namespace**, который пользователь задаёт флагом: Рисунок (11)

```
clowixdev@clowixdev ~> unshare -h

Usage:
  unshare [options] [<program> [<argument>...]]

Run a program with some namespaces unshared from the parent.

Options:
  -m, --mount[=<file>]      unshare mounts namespace
  -u, --uts[=<file>]        unshare UTS namespace (hostname etc)
  -i, --ipc[=<file>]        unshare System V IPC namespace
  -n, --net[=<file>]        unshare network namespace
  -p, --pid[=<file>]        unshare pid namespace
  -U, --user[=<file>]        unshare user namespace
  -C, --cgroup[=<file>]     unshare cgroup namespace
  -T, --time[=<file>]       unshare time namespace
```

Рисунок 11 – синтаксис и доступные флаги команды **unshare**

Таким образом, не прибегая к самописным решениям и установке каких либо дополнительных утилит, можно запустить нужный пользователю процесс в отдельном от родителя namespace.

### 3.2. clone(...)

Системный вызов **clone()** можно использовать с некоторыми флагами, при указании которых в аргументах функции, создадутся определённые пространства имён:

- **CLONE\_NEWNS** – новое пространство имен MNT
- **CLONE\_NEWUTS** – новое пространство имен UTS
- **CLONE\_NEWIPC** – новое пространство имен IPC
- **CLONE\_NEWPID** – новое пространство имен PID
- **CLONE\_NEWNET** – новое пространство имен NET
- **CLONE\_NEWUSER** – новое пространство имен USR

Разница между **clone** и **unshare** в том, что первый порождает новый

процесс внутри нового набора пространств имен, а последний перемещает в новый набор пространств имен текущий процесс.

#### 4. Примеры использования пространств имён.

Пространство имён даёт процессам, запущенным в контейнерах, иллюзию, что они имеют свои собственные ресурсы. Основная цель изоляции процессов состоит в предотвращении вмешательства процессов одного контейнера в работу других контейнеров, а также работу хостовой машины.

На данный момент существует несколько технологий контейнеризации, основными же являются **LXC** и **Docker**. И в основе этих технологиях лежат пространства имён и они имеют одинаковый принцип работы.

При создании контейнера, инструмент создаёт и настраивает новые пространства имён и работает в них, для полной изоляции контейнера и защиты системных ресурсов хостовой ОС.

#### 5. Процесс создания нового пространства имён.

Рассмотрим процесс создания **PID namespace** с точки зрения действий, которые выполняет Linux.

Изначально, в ОС Linux, все процессы объединены в дерево, с корнем-процессом с **PID 1**. При создании нового **PID namespace**, определяется поддерево, внутри которого, созданный дочерний процесс имеет **PID 1**, относительно своего поддерева и является там родительским для всех новых процессов (Рисунок 12).

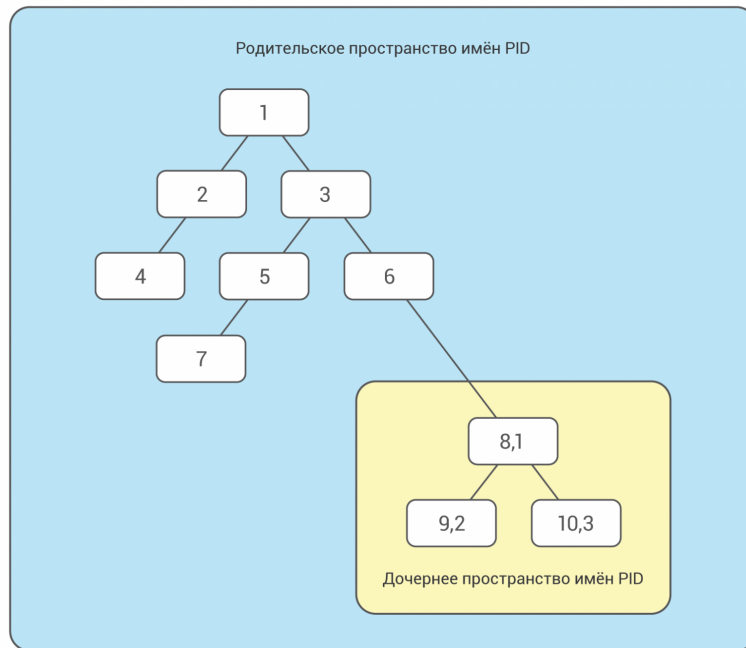


Рисунок 12 – дерево процессов в ОС Linux

Рассмотрим структуру, которая определяет пространство имён PID в исходных файлах ядра Linux: (Листинг 1)

#### Листинг 1 – Структура PID namespace

```

struct pid_namespace {
    struct idr idr;
    struct rcu_head rcu;
    unsigned int pid_allocated;
    struct task_struct *child_reaper;
    struct kmem_cache *pid_cache;
    unsigned int level;
    int pid_max;
    struct pid_namespace *parent;
#ifdef CONFIG_BSD_PROCESS_ACCT
    struct fs_pin *bacct;
#endif
    struct user_namespace *user_ns;
    struct ucounts *ucounts;
    int reboot; /* group exit code if this pidns was rebooted */
    struct ns_common ns;
    struct work_struct work;
#ifdef CONFIG_SYSCTL
    struct ctl_table_set set;
    struct ctl_table_header *sysctls;
#endif
#ifdef CONFIG_MEMFD_CREATE
    int memfd_noexec_scope;
#endif
}

```

Далее рассмотрим функцию ядра, отвечающую за создание нового

## PID namespace – create\_pid\_namespace(...) (Листинг 2)

### Листинг 2 – функция создания PID namespace

```
static struct pid_namespace *create_pid_namespace(struct user_namespace
*user_ns,
    struct pid_namespace *parent_pid_ns)
{
    struct pid_namespace *ns;
    unsigned int level = parent_pid_ns->level + 1;
    struct ucounts *ucounts;
    int err;

    err = -EINVAL;
    if (!in_userns(parent_pid_ns->user_ns, user_ns))
        goto out;

    err = -ENOSPC;
    if (level > MAX_PID_NS_LEVEL)
        goto out;
    ucounts = inc_pid_namespaces(user_ns);
    if (!ucounts)
        goto out;

    err = -ENOMEM;
    ns = kmem_cache_zalloc(pid_ns_cachep, GFP_KERNEL);
    if (ns == NULL)
        goto out_dec;

    idr_init(&ns->idr);

    ns->pid_cachep = create_pid_cachep(level);
    if (ns->pid_cachep == NULL)
        goto out_free_idr;

    err = ns_alloc_inum(&ns->ns);
    if (err)
        goto out_free_idr;
    ns->ns.ops = &pidns_operations;

    ns->pid_max = PID_MAX_LIMIT;
    err = register_pidns_sysctls(ns);
    if (err)
        goto out_free_inum;

    refcount_set(&ns->ns.count, 1);
    ns->level = level;
    ns->parent = get_pid_ns(parent_pid_ns);
    ns->user_ns = get_user_ns(user_ns);
    ns->ucounts = ucounts;
    ns->pid_allocated = PIDNS_ADDING;
    INIT_WORK(&ns->work, destroy_pid_namespace_work);

#ifdef CONFIG_SYSCTL && defined(CONFIG_MEMFD_CREATE)
    ns->memfd_noexec_scope = pidns_memfd_noexec_scope(parent_pid_ns);
#endif

    return ns;

out_free_inum:
    ns_free_inum(&ns->ns);
}
```

```
out_free_idr:
    idr_destroy(&ns->idr);
    kmem_cache_free(pid_ns_cache, ns);
out_dec:
    dec_pid_namespaces(ucounts);
out:
    return ERR_PTR(err);
}
```

Как можно заметить, при создании процесса проводятся различные проверки, на предмет невозможности создания нового пространства имён, например, проверка на текущий уровень вложенности, который не должен превышать 32.

Далее аллоцируется необходимое пространство и устанавливаются значения полей структуры **namespace**, например **PID** родительского **namespace** и уровень вложенности.

После всех проведённых настроек, если не произошло никаких ошибок, созданная структура возвращается и создание нового **namespace** на этом завершается.



## ПРАКТИЧЕСКАЯ ЧАСТЬ

### Разработка программы контейнеризатора на C++.

В ходе работы была создана программа, которая позволяет запускать в изолированном пространстве имён. Консоль имеет **PID 1**, видит уникальное дерево монтирования и изменения, выполненные внутри контейнера не влияют на хост-систему. (Рисунок 13)

```
clowixdev@clowixdev /m/d/a/Y/l/p/c/o/coursework (main)> ./bin/isolate sh
Launching <sh> in isolated namespaces...

-----
/ # ps
PID    USER      TIME  COMMAND
   1   root         0:00  sh
   2   root         0:00  ps
/ # ls
bin      etc       lib       mnt       proc      run       srv       tmp       var
dev      home     media     opt       root      sbin      sys      usr
/ # ls -l | head -n 2
total 0
drwxrwxrwx    1 root    root           4096 May 30 12:13 bin
/ # hostname
clowixdev
/ # hostname changed_test_hostname
/ # hostname
changed_test_hostname
/ # exit
clowixdev@clowixdev /m/d/a/Y/l/p/c/o/coursework (main)> █
```

Рисунок 13 – выполнение команд внутри контейнера

Далее, после завершения работы в контейнере, были выполнены такие же команды: (Рисунок 14)

```
clowixdev@clowixdev /m/d/a/Y/l/p/c/o/coursework (main)> ps
  PID TTY          TIME CMD
  686 pts/0        00:00:07 fish
 10878 pts/0        00:00:00 ps
clowixdev@clowixdev /m/d/a/Y/l/p/c/o/coursework (main)> ls /
bin  change_meminfo  etc  init  lib32  libx32  media  opt  root  sbin  srv  tmp  var
boot dev           home  lib   lib64  lost+found  mnt   proc  run  snap  sys  usr
clowixdev@clowixdev /m/d/a/Y/l/p/c/o/coursework (main)> ls -l / | head -n 2
total 2740
lrwxrwxrwx    1 root    root           7 May 31  2023 bin -> usr/bin
clowixdev@clowixdev /m/d/a/Y/l/p/c/o/coursework (main)> hostname
clowixdev
clowixdev@clowixdev /m/d/a/Y/l/p/c/o/coursework (main)> █
```

Рисунок 14 – выполнение команд в хост-системе

Была выполнена изоляция в **UTS**, **USER**, **PID** и **MNT** пространствах имён, с помощью системного вызова **clone(...)**, с соответствующими флагами - **CLONE\_NEWUTS**, **CLONE\_NEWUSER**, **CLONE\_NEWNS** и **CLONE\_NEWPID**.

Опишем процесс изоляции для каждого из пространства имён.

## 1. UTS namespace

Для изоляции процесса в отношении пространства имён **UTS** потребовалось только указать соответствующий флаг **CLONE\_NEWUTS**. (Рисунок 15)

```
clowixdev@clowixdev /m/d/a/Y/l/p/c/o/coursework (main)> hostname
clowixdev
clowixdev@clowixdev /m/d/a/Y/l/p/c/o/coursework (main)> ./bin/isolate sh
Launching <sh> in isolated namespaces...
-----
/ # hostname
clowixdev
/ # hostname changed_test_hostname
/ # hostname
changed_test_hostname
/ # exit
clowixdev@clowixdev /m/d/a/Y/l/p/c/o/coursework (main)> hostname
clowixdev
clowixdev@clowixdev /m/d/a/Y/l/p/c/o/coursework (main)> █
```

Рисунок 15 – демонстрация изменения hostname

## 2. USER namespace

При указание флага во время создании дочернего процесса, процесс в новом пространстве имён не имел корректного **UID** и **GID** (Рисунок 16), что могло вызывать ошибки при проведении проверок разрешений процесса и фатально сказаться на безопасности хост-системы.

```

clowixdev@clowixdev /m/d/a/Y/l/p/c/o/coursework (main)> ./bin/isolate sh
Launching <sh> in isolated namespaces...

-----
/ $ ls -l | head -n 5
total 0
drwxrwxrwx    1 nobody   nobody           4096 May 30 12:13 bin
drwxrwxrwx    1 nobody   nobody           4096 May 30 12:13 dev
drwxrwxrwx    1 nobody   nobody           4096 May 30 12:13 etc
drwxrwxrwx    1 nobody   nobody           4096 May 30 12:13 home
/ $ █

```

Рисунок 16 – **UID** и **GID** нового процесса

Для корректной изоляции процесса в отношении **USER namespaces**, потребовалось дополнительно внести изменения в некоторые файлы, а именно в файлы файловой системы, отвечающие за маппинг **ID** пользователей и групп.

Разработанная функция **setup\_uns(...)** (Листинг 3), отвечает за предварительную подготовку **map-файлов** хост-системы, а именно **/proc/<pid>/uid\_map** и **/proc/<pid>/gid\_map**, к созданию нового пространства имён **USER**, внося необходимые изменения, для представления пользователя в новом namespace как **ID 0**, т.е. как пользователя **root**. (Рисунок 17)

Листинг 3 – разработанная функция **setup\_uns(...)**

```

void setup_uns(int pid) {
    stringstream path;
    stringstream line;

    path << "/proc/" << pid << "/uid_map";
    line << "0 " << DEFAULT_UID << " 1\n";
    ofstream uid_mapper(path.str());
    uid_mapper << line.str();
    uid_mapper.close();

    clean_ss(path, line);

    path << "/proc/" << pid << "/setgroups";
    line << "deny";
    ofstream setgroups_setup(path.str());
    setgroups_setup << line.str();
    setgroups_setup.close();

    clean_ss(path, line);

    path << "/proc/" << pid << "/gid map";

```

```
line << "0 " << DEFAULT_UID << " 1\n";
ofstream gid_mapper(path.str());
gid_mapper << line.str();
gid_mapper.close();
}
```

```
clowixdev@clowixdev /m/d/a/Y/l/p/c/o/coursework (main)> ./bin/isolate sh
Launching <sh> in isolated namespaces...

-----
/ # ls -l | head -n 5
total 0
drwxrwxrwx    1 root    root    4096 May 30 12:13 bin
drwxrwxrwx    1 root    root    4096 May 30 12:13 dev
drwxrwxrwx    1 root    root    4096 May 30 12:13 etc
drwxrwxrwx    1 root    root    4096 May 30 12:13 home
/ # █
```

Рисунок 17 – **UID** и **GID** нового процесса после корректировок

Хоть в новом **USER namespace**, пользователь и имеет **ID 0**, и, как кажется, все права суперпользователя, но все его права остаются на уровне родителя. Происходит это из-за древовидной структуры пространств имён. Поэтому при создании нового **USER namespace**, пользователь контейнера является **ID 0**, но не имеет прав суперпользователя по отношению к хостовой ОС, так как является потомком процесса «изолятора»

### 3. **MNT namespace**

Для изоляции процесса в отношении пространства имён **MNT**, рассмотрим принцип его работы.

Среди файлов в ОС Linux, есть файл, который содержит в себе все точки монтирования системы. Это файл **/proc/<pid>/mounts**. При создании нового **MNT namespace**, Linux делает копию этого файла, поэтому дочерний процесс хоть и выполняется в новом пространстве имён, но всё так же имеет доступ и знает о всех файлах ОС, а так как ядро Linux только создаёт ссылки, а не копии всех файлов, то изменения в этих файлах будут напрямую влиять на копии этих файлов в других namespaces.

Таким образом было принято решение собирать файловую систему «песочницу» с минимальным набором файлов и утилит для корректной работы дочернего процесса.

Такая файловая система, достаточная для работы «из коробки», была взята у **Alpine Linux** (Рисунок 18)

## DOWNLOADS

Current Alpine Version **3.22.0** (Released May 30, 2025)

 GPG 0482 D840 22F5 2DF1 C4E7 CD43 293A CD09 07D9 495A

 [Download or Launch Cloud Images](#)

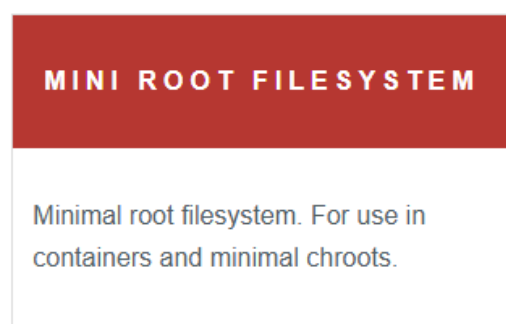


Рисунок 18 – мини-файловая система для «песочницы»

Далее была разработана функция **setup\_mns(...)** (Листинг 4), которая последовательно монтирует скачанную файловую систему во временную папку, затем, использует системный вызов **pivot\_root**.

Листинг 4 – разработанная функция **setup\_mns(...)**

```
void setup_mns(const char *rootfs) {
    const char *mnt = rootfs;

    if (mount(rootfs, mnt, "ext4", MS_BIND, "")) {
        ...
    }

    if (chdir(mnt)) {
        ...
    }

    const char *old_fs = ".old_fs";

    if (mkdir(old_fs, 0777) && errno != EEXIST) {
```

```

    ...
}

if (syscall(SYS_pivot_root, ".", old_fs)) {
    ...
}

if (chdir("/")) {
    ...
}

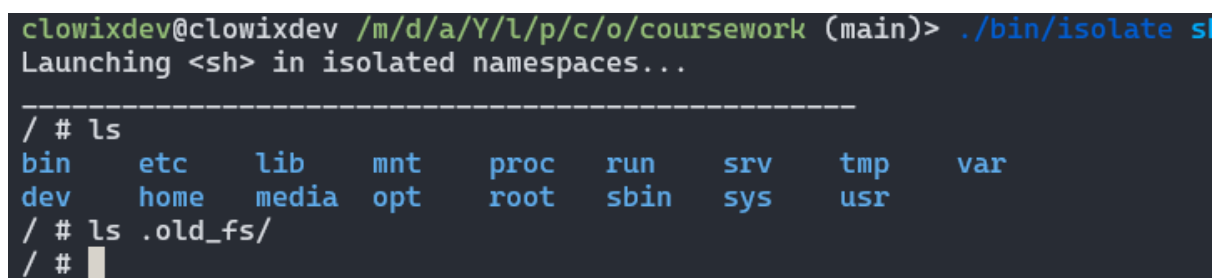
...

if (umount2(old_fs, MNT_DETACH)) {
    ...
}
}

```

Команда принимает два аргумента: **pivot\_root new\_root put\_old**, где **new\_root** — это путь к файловой системе, будущей вскоре корневой файловой системой, а **put\_old** — путь к каталогу, куда в новой файловой системе будет помещена старая корневая.

После исполнения системного вызова и замены корневого каталога, можно размонтировать старую файловую систему, которая находится по пути **put\_old**. Таким образом, дочерний процесс не будет ничего знать о хостовой файловой системе, так как у него будет своя, без доступа к хосту. (Рисунок 19)



```

clowixdev@clowixdev /m/d/a/Y/l/p/c/o/coursework (main)> ./bin/isolate s
Launching <sh> in isolated namespaces...

-----
/ # ls
bin    etc    lib    mnt    proc   run    srv    tmp    var
dev    home   media  opt    root   sbin   sys    usr
/ # ls .old_fs/
/ # █

```

Рисунок 19 – корневой каталог дочернего процесса

#### 4. PID namespace

Для создания нового пространства имён **PID** указывается соответствующий флаг при создании дочернего процесса. Однако при

проверке **PID** у процесса с помощью утилиты **ps**, ничего узнать не получится (Рисунок 20)

```
clowixdev@clowixdev /m/d/a/Y/l/p/c/o/coursework (main)> ./bin/isolate sh
Launching <sh> in isolated namespaces...

-----
/ # ps
PID  USER      TIME  COMMAND
/ # █
```

Рисунок 20 – проверка **PID** процесса

Так происходит потому что для корректной работы, системе Linux нужна специальная файловая система **proc**, в файлах которой и содержатся все сведения о системе. Для того чтобы установить её, необходимо лишь сообщить Linux о том, что она нам нужна и в каком месте её смонтировать.

Была модифицирована функция **setup\_mns(...)**. В неё было добавлено монтирование файловой системы **proc** (Листинг 5)

Листинг 5 – монтирование **proc**

```
void setup_mns(const char *rootfs) {
    ...

    if (mkdir("/proc", 0555) && errno != EEXIST) {
        ...
    }

    if (mount("proc", "/proc", "proc", 0, "")) {
        ...
    }

    ...
}
```

После этого, можно увидеть, что процесс запущен с **PID 1**. (Рисунок 21)

```
clowixdev@clowixdev /m/d/a/Y/l/p/c/o/coursework (main)> ./bin/isolate sh
Launching <sh> in isolated namespaces...

-----
/ # ps
PID    USER      TIME  COMMAND
   1   root         0:00   sh
   2   root         0:00   ps
/ # █
```

Рисунок 21 – **PID** запущенного процесса.



## ЗАКЛЮЧЕНИЕ

В ходе выполнения курсовой работы была изучена технология **Linux namespaces**, а также разработана программа-контейнеризатор, использующая технологию пространств имён для изоляции запущенного в ней процесса.

```

/*
Isolate - containerization utility
Mishenev Nikita 5131001/30002

*/

#include <stdarg.h>
#include <sys/prctl.h>
#include <wait.h>
#include <sys/mount.h>
#include <sys/stat.h>
#include <syscall.h>

#include <iostream>
#include <fstream>
#include <sstream>

using namespace std;

#define STACKSIZE      (1024*1024)
#define DEFAULT_UID    (1000)

static char cmd_stack[STACKSIZE];

void setup_mns(const char *);
void prepare_procfs(void);

struct params {
    int fd[2];
    char **argv;
};

void display_help() {
    cout << "Usage: isolate [COMMAND]...\n"
           "Run command in isolated space\n\n"
           "\t-h, --help\t\tdisplay this help message and exit\n\n"
           "Examples:\n"
           "\tisolate bash\t\ttruns bash console in isolated space\n"
           "\tisolate ./test_script\ttruns your script in isolated space\n";
}

template <typename T, typename...Params>
void kill_thread(T estr, Params&&...params) {
    printf(estr, params...);

    exit(1);
}

void parse_args(int argc, char **argv, params *params) {
    if (--argc < 1) {
        display_help();
        exit(0);
    }

    params->argv = ++argv;
}

void await_setup(int pipe) {
    char buf[2];
    if (read(pipe, buf, 2) != 2) {
        kill_thread("Failed to read from pipe: %m\n");
    }
}

int cmd_exec(void *arg) {
    if (prctl(PR_SET_PDEATHSIG, SIGKILL)) {
        kill_thread("cannot PR_SET_PDEATHSIG for child process: %m\n");
    }
}

```

```

params *params = (struct params *)arg;
await_setup(params->fd[0]);

setup_mns("mini_rootfs");

if (setgid(0) == -1) {
    kill_thread("Failed to setgid: %m\n");
}
if (setuid(0) == -1) {
    kill_thread("Failed to setuid: %m\n");
}

char **argv = params->argv;
char *cmd = argv[0];
cout << "Launching <"<< cmd << "> in isolated namespaces...\n"
      "_____ " << endl;

if (execvp(cmd, argv) == -1) {
    kill_thread("Failed to exec %s: %m\n", cmd);
}

return 1;
}

void clean_ss(stringstream &ss1, stringstream &ss2) {
    ss1.str("");
    ss1.clear();
    ss2.str("");
    ss2.clear();
}

void setup_uns(int pid) {
    stringstream path;
    stringstream line;

    path << "/proc/" << pid << "/uid_map";
    line << "0 " << DEFAULT_UID << " 1\n";
    ofstream uid_mapper(path.str());
    uid_mapper << line.str();
    uid_mapper.close();

    clean_ss(path, line);

    path << "/proc/" << pid << "/setgroups";
    line << "deny";
    ofstream setgroups_setup(path.str());
    setgroups_setup << line.str();
    setgroups_setup.close();

    clean_ss(path, line);

    path << "/proc/" << pid << "/gid_map";
    line << "0 " << DEFAULT_UID << " 1\n";
    ofstream gid_mapper(path.str());
    gid_mapper << line.str();
    gid_mapper.close();
}

void setup_mns(const char *rootfs) {
    const char *mnt = rootfs;

    if (mount(rootfs, mnt, "ext4", MS_BIND, "")) {
        kill_thread("Failed to mount %s at %s: %m\n", rootfs, mnt);
    }

    if (chdir(mnt)) {
        kill_thread("Failed to chdir to rootfs mounted at %s: %m\n", mnt);
    }

    const char *old_fs = ".old_fs";

    if (mkdir(old_fs, 0777) && errno != EEXIST) {
        kill_thread("Failed to mkdir old_fs %s: %m\n", old_fs);
    }

    if (syscall(SYS_pivot_root, ".", old_fs)) {
        kill_thread("Failed to pivot root from %s to %s: %m\n", rootfs, old_fs);
    }
}

```

```

    }

    if (chdir("/")) {
        kill_thread("Failed to chdir to new root: %m\n");
    }

    if (mkdir("/proc", 0555) && errno != EEXIST) {
        kill_thread("Failed to mkdir /proc: %m\n");
    }

    if (mount("proc", "/proc", "proc", 0, "")) {
        kill_thread("Failed to mount proc: %m\n");
    }

    if (umount2(old_fs, MNT_DETACH)) {
        kill_thread("Failed to umount old_fs %s: %m\n", old_fs);
    }
}

int main(int argc, char **argv) {
    params params = {0, 0};
    parse_args(argc, argv, &params);

    if (pipe(params.fd) < 0) {
        kill_thread("Failed to create pipe: %m");
    }

    int clone_flags = SIGCHLD | CLONE_NEWUTS | CLONE_NEWUSER | CLONE_NEWNS | CLONE_NEWPID;
    int cmd_pid = clone(cmd_exec, cmd_stack + STACKSIZE, clone_flags, &params);

    if (cmd_pid < 0) {
        kill_thread("Failed to clone: %m\n");
    }

    int pipe = params.fd[1];

    setup_uns(cmd_pid);

    if (write(pipe, "OK", 2) != 2) {
        kill_thread("Failed to write to pipe: %m");
    }

    if (close(pipe)) {
        kill_thread("Failed to close pipe: %m");
    }

    if (waitpid(cmd_pid, NULL, 0) == -1) {
        kill_thread("Failed to wait pid %d: %m\n", cmd_pid);
    }

    return 0;
}

```