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**

**К У Р С О В А Я Р А Б О Т А**

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](#__RefHeading___1)

[ОСНОВНАЯ ЧАСТЬ 4](#__RefHeading___2)

[ТЕОРЕТИЧЕСКАЯ ЧАСТЬ 4](#__RefHeading___3)

[1. Пространство имён или namespaces. 4](#__RefHeading___4)

[2. Типы пространств имен в Linux. 5](#__RefHeading___5)

[3. Создание пространств имён. 11](#__RefHeading___6)

[3.1. Unshare 11](#__RefHeading___7)

[3.2. clone(...) 12](#__RefHeading___8)

[4. Примеры использования пространств имён. 13](#__RefHeading___9)

[ПРАКТИЧЕСКАЯ ЧАСТЬ 14](#__RefHeading___10)

[Разработка программы контейнеризатора на С++. 14](#__RefHeading___11)

[1. UTS namespace 15](#__RefHeading___12)

[2. USER namespace 15](#__RefHeading___18)

[3. MNT namespace 17](#__RefHeading___19)

[4. PID namespace 19](#__RefHeading___20)

[ЗАКЛЮЧЕНИЕ 22](#__RefHeading___16)

[ПРИЛОЖЕНИЕ А. ИСХОДНЫЙ КОД РАЗРАБОТАННОГО ПРИЛОЖЕНИЯ. 23](#__RefHeading___17)

# 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 С, которая запускает процесс в отдельных *namespaces.*
* Сделать выводы о проделанной работе

# ОСНОВНАЯ ЧАСТЬ

## **ТЕОРЕТИЧЕСКАЯ ЧАСТЬ**

## 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)

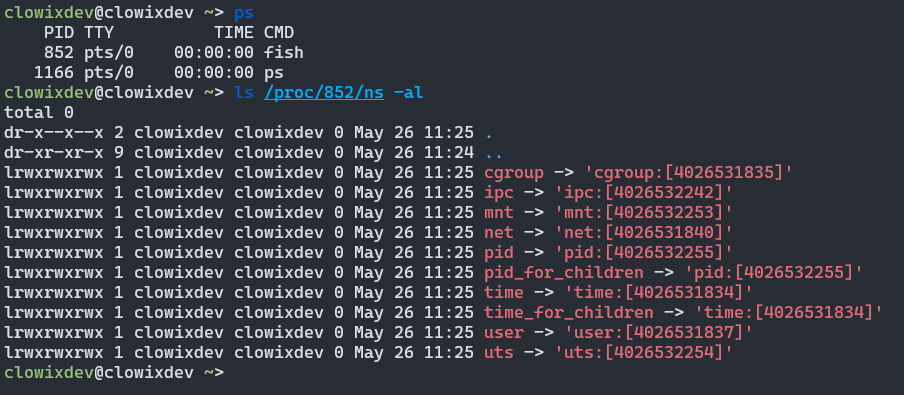


Рисунок 1 – пространство имён для запущенного терминала

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

## 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:

* 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.

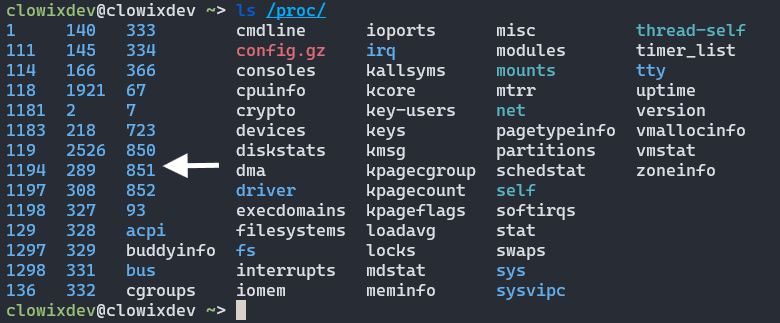


Рисунок 2 – файловая система **procfs**

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:

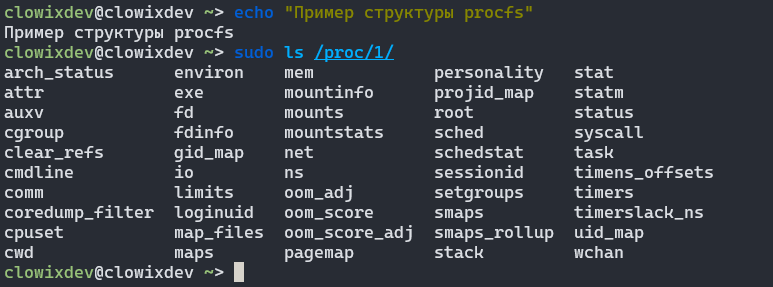


Рисунок 3 – пример структуры **procfs** процесса с **PID 1**

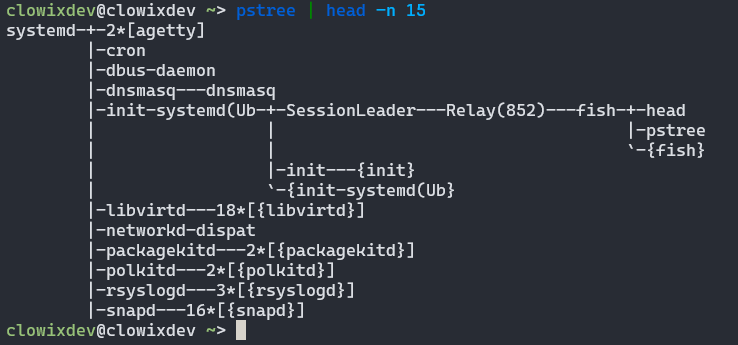


Рисунок 4 – пример структуры дерева процессов.

* 1. **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)

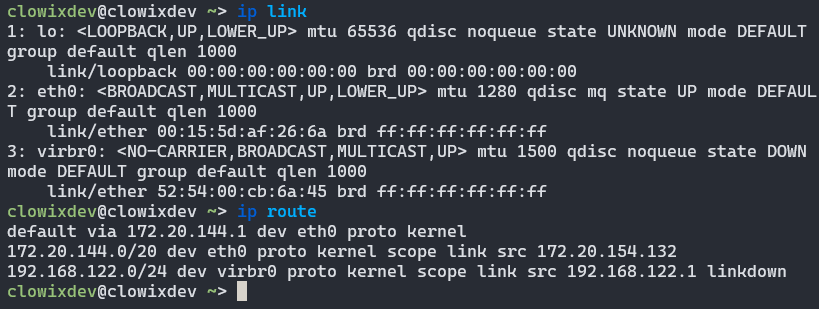


Рисунок 5 – пример сетевых интерфейсов и правил маршрутизации

* 1. **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)

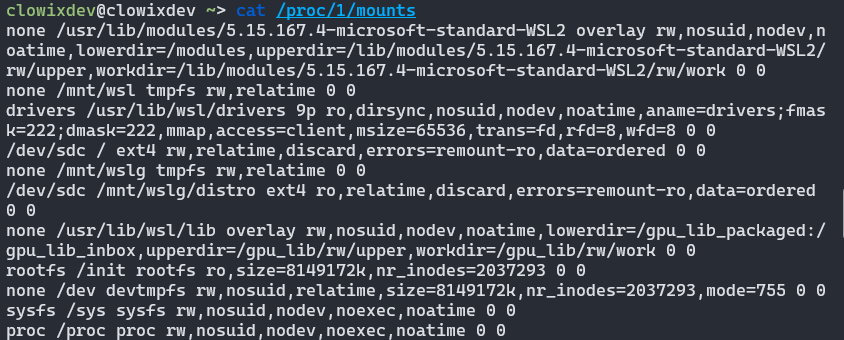


Рисунок 6 – **MNT** структурадля процесса с **PID 1**

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)

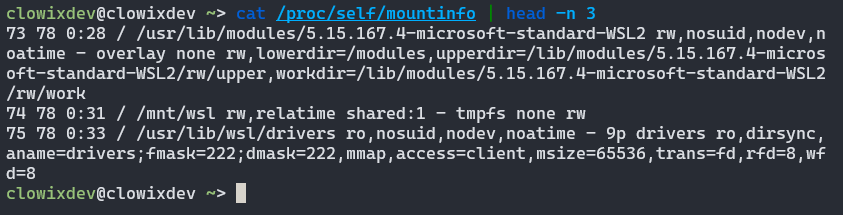


Рисунок 7 – список точек монтирования в namespace текущего процесса

* 1. **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)

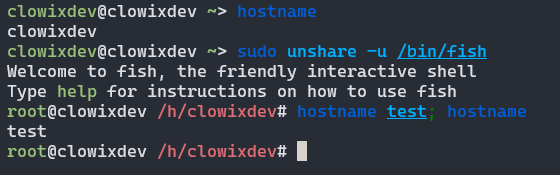


Рисунок 8 – изменение **hostname** в другом **UTS** **namespace**

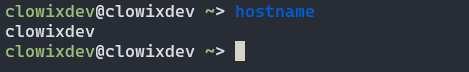


Рисунок 9 – проверка **hostname** в другом терминале

* 1. **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:

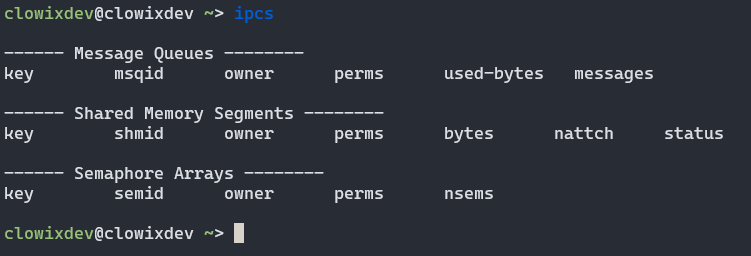


Рисунок 10 – пример IPC namespace для текущего процесса

* 1. **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)

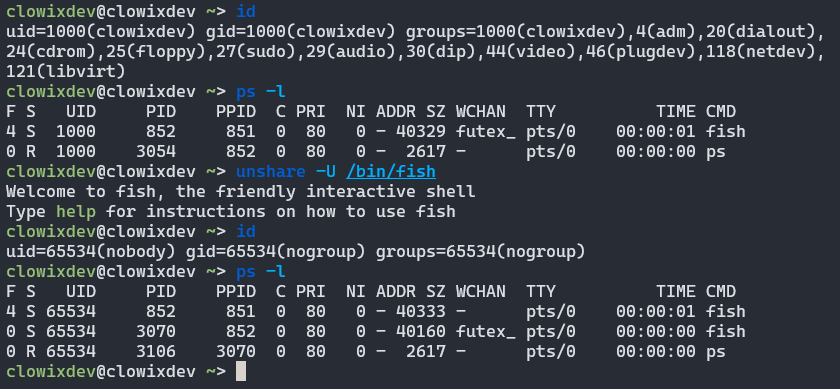


Рисунок 11 – пример создания нового пользовательского пространства имён

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.

## Создание пространств имён.

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

## Unshare

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

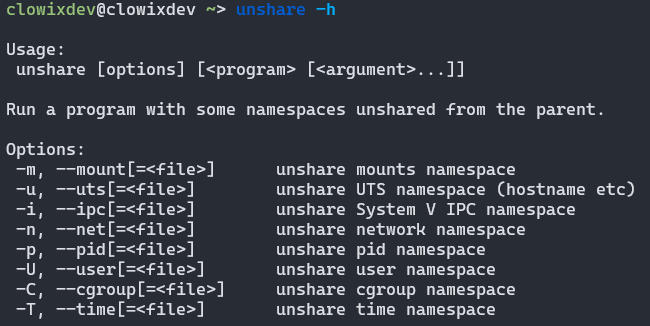


Рисунок 11 – синтаксис и доступные флаги команды **unshare**

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

## 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.

## 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.

## ПРАКТИЧЕСКАЯ ЧАСТЬ

### Разработка программы контейнеризатора на С++.

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 12)

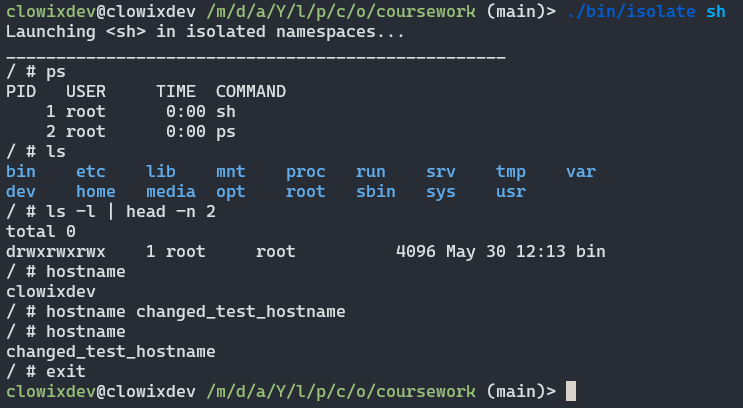


Рисунок 12 – выполнение команд внутри контейнера

Далее, после завершения работы в контейнере, были выполнены такие же команды: (Рисунок 13)

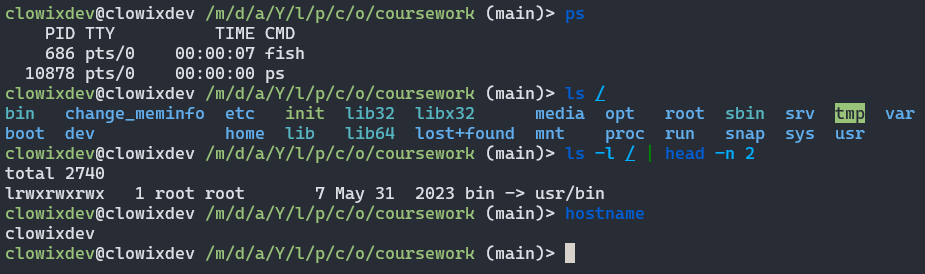


Рисунок 13 – выполнение команд в хост-системе

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.

### UTS namespace

Для изоляции процесса в отношении пространства имён **UTS** потребовалось только указать соответствующий флаг **CLONE\_NEWUTS**. (Рисунок 14)

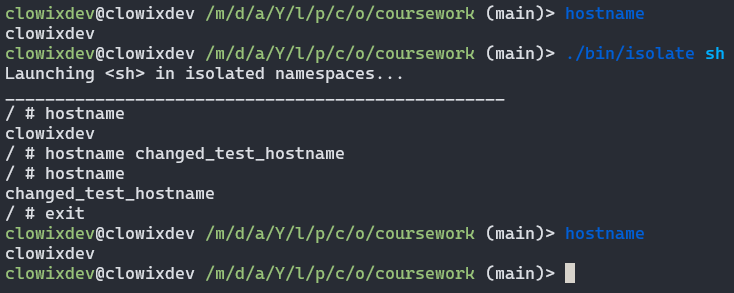


Рисунок 14 – демонстрация изменения hostname

### USER namespace

При указание флага во время создании дочернего процесса, процесс в новом пространстве имён не имел корректного **UID** и **GID** (Рисунок 15), что могло вызывать ошибки при проведении проверок разрешений процесса и фатально сказаться на безопасности хост-системы.

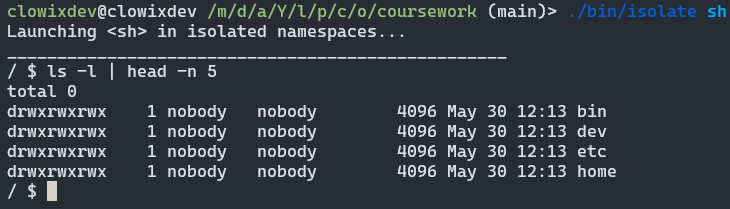


Рисунок 15 – **UID** и **GID** нового процесса

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 1)**,** 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 16)

Листинг 1 – 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();  } |

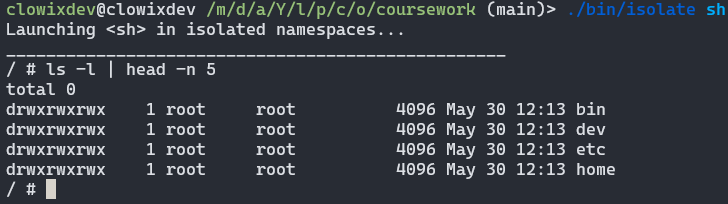


Рисунок 16 – **UID** и **GID** нового процесса после корректировок

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»

### 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 17)

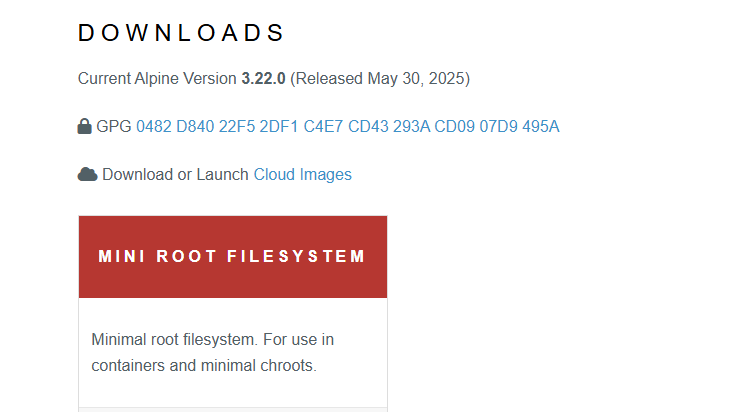


Рисунок 17 – мини-файловая система для «песочницы»

Далее была разработана функция **setup\_mns(...)** (Листинг 2)**,** которая последовательно монтирует скачанную файловую систему во временную папку, затем,использует системный вызов **pivot\_root**.

Листинг 2 – 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 18)

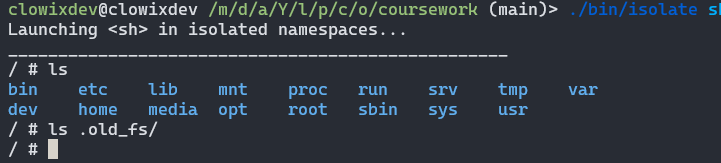


Рисунок 18 – корневой каталог дочернего процесса

### 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 19)

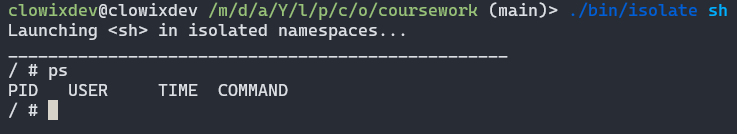


Рисунок 19 – проверка **PID** процесса

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 3)

Листинг 3 – 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 20)

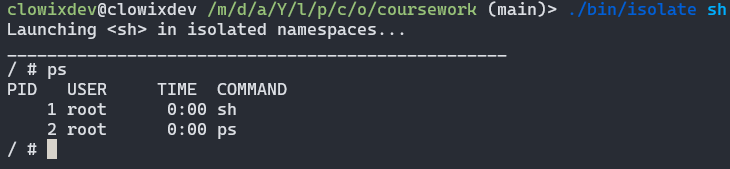


Рисунок 20 – **PID** запущенного процесса.

# 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.

# ПРИЛОЖЕНИЕ А. ИСХОДНЫЙ КОД РАЗРАБОТАННОГО ПРИЛОЖЕНИЯ.

|  |
| --- |
| /\*  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\truns bash console in isolated space\n"  "\tisolate ./test\_script\truns 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;  } |