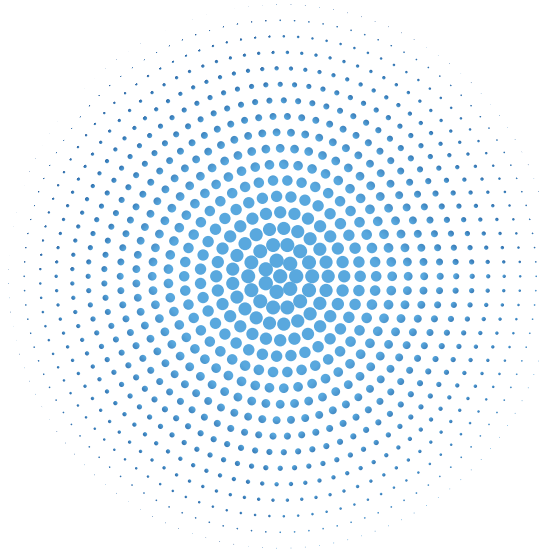
МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ЯДЕРНЫЙ УНИВЕРСИТЕТ «МИФИ»



Д.В. ЕФАНОВ

БАЗОВЫЕ МЕХАНИЗМЫ ЗАЩИТЫ ЯДРА LINUX

Учебное пособие

Международный научно-методический центр
Москва 2022

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ЯДЕРНЫЙ
УНИВЕРСИТЕТ «МИФИ»

Д.В. Ефанов

Базовые механизмы защиты ядра Linux

Учебное пособие

Москва 2022

УДК 004.451.056(075)
ББК 32.973.26-012.2я7
Е90

Ефанов Д.В. Базовые механизмы защиты ядра Linux: Учебное пособие. [Электронный ресурс]. М.: НИЯУ МИФИ, 2022. – 192 с.

Учебное пособие посвящено рассмотрению основных возможностей ядра Linux, обеспечивающих безопасность функционирования операционной системы и прикладных программ. Особенностью пособия является изучение средств защиты на трех уровнях: командного интерфейса, системных вызовов, а также на уровне структур данных и алгоритмов ядра, что позволяет создать у обучающихся целостное восприятие работы механизмов защиты ядра Linux.

Пособие предназначено для студентов высших учебных заведений, обучающихся по программам бакалавриата, магистратуры и специалитета укрупненного направления 10.00.00 «Информационная безопасность».

Подготовлено в рамках Проекта по созданию и развитию Международного научно-методического центра НИЯУ МИФИ.

Рецензент канд. физ.-мат. наук, вед. науч. сотр.
Института системного программирования им. В.П.Иванникова РАН
А.В. Хорошилов

Рекомендовано к изданию кафедрой криптологии и кибербезопасности
(№ 42) НИЯУ МИФИ

ISBN 978-5-7262-2924-9

Оглавление

Предисловие

Двадцать лет назад вышло учебное пособие «Алгоритмы и структуры ядра Linux» [1], в котором были обобщены результаты работы научной группы под руководством доцента кафедры 12 «Компьютерные системы и технологии» В.Д. Никитина.

Это были времена, когда Linux не был широко известен, но на кафедре 12, благодаря заведующему кафедрой Л.Д. Забродину, большое внимание уделялось изучению операционной системы UNIX System V, что создавало условия для освоения Linux.

Тогда включение Linux в учебный процесс было обусловлено его свободным распространением и возможностью изучения его алгоритмов на уровне исходного кода. Однако, как показало время, ядро Linux вместе с программами проекта GNU стали самой распространенной операционной системой в мире! Таким образом, еще в конце 90-х на кафедре 12 удалось угадать тренд и наполнить учебный процесс новейшими знаниями, определившими направления работ на много лет вперед.

Сегодня системы на основе GNU/Linux применяются во многих критически важных областях: в научных экспериментах, в военных, банковских, медицинских и промышленных автоматизированных системах. *Владение Linux как технологией стало необходимым элементом обеспечения цифрового суверенитета государства.*

Но GNU/Linux – это только набор программ, и чтобы их правильно использовать, нужны грамотные инженеры, а также последовательная многолетняя подготовка кадров. Данное учебное пособие призвано помочь студентам быстрее войти в мир GNU/Linux и создать основу для более глубокого его изучения.

Введение

Настоящее пособие предназначено для студентов, обучающихся по программам бакалавриата, магистратуры и специалитета укрупненного направления 10.00.00 «Информационная безопасность».

В первой главе рассмотрены основные понятия, общая концепция защиты операционной системы, а также базовые средства защиты, на которых строится работа всей операционной системы.

В главах 2, 3 и 4 рассмотрены две основные сущности операционной системы: файлы и процессы, досконального знание которых является обязательным для будущих специалистов по информационной безопасности.

В пятой главе отдельно рассмотрена система мандатного управления доступом, без которой трудно представить современную операционную систему.

В шестой главе рассказывается о том, как организовать работу пользователей системы.

В списке литературы перечислены книги, статьи и нормативные документы, которые использовались при подготовке пособия.

Учебное пособие включает многочисленные примеры программ на языках C и BASH. Примеры, заимствованные из [9], снабжены указанием на листинг из книги. Примеры, заимствованные из справочной системы **man**, снабжены указанием на соответствующие справочные страницы. Исходный код примеров можно скачать из репозитория https://github.com/efanov/linux-textbook.

Пособие может быть использовано в курсе по безопасности операционных систем, а также для самостоятельного изучения современных средств защиты операционной системы GNU/Linux.

Ядро Linux – пример успешного, самого масштабного проекта с открытым исходным кодом. Одновременно над ядром работает около 1500 разработчиков, разбросанных по всему миру.

Ядро Linux давно перестало быть проектом только энтузиастов – сегодня 80 % разработчиков финансируются примерно 250 компаниями, связавшими свой бизнес с GNU/Linux.

Благодаря чему в нашем распоряжении появился такой мощный инструмент, как Linux? Ответ заключается в природе Linux как явлении. Linux – свободное программное обеспечение, что позволяет

использовать, изучать и дорабатывать его под свои нужды. Успешная судьба Linux была определена задолго до его рождения появлением проекта GNU по разработке полностью открытой и доступной UNIX-подобной операционной системы.

Основные механизмы защиты сосредоточены в ядре Linux. Среди релизов ядра можно выделить так называемые longterm-релизы, которые характеризуются долгим периодом исправления ошибок (bugfixes). Представленные в пособии структуры ядра относятся к версии 5.10, которая была выпущена в декабре 2020 г. и будет поддерживаться до конца 2026 г. Информацию о версиях ядра Linux можно получить на сайте https://www.kernel.org/. Разработчики конкретного дистрибутива GNU/Linux могут использовать другую версию ядра для долговременной поддержки. Узнать текущую версию ядра можно с помощью команды **uname -r**. Исходные коды ядра Linux можно скачать из git-репозитория Линуса Торвальдса по адресу https://github.com/torvalds/linux или из репозитория установленного дистрибутива GNU/Linux. А читать исходные коды удобно в специальных справочных системах, например по адресу https://elixir.bootlin.com/linux/v5.10/source.

Глава 1. Концепция защиты

В данной главе вводятся основные понятия и описываются базовые компоненты, на которых строится защита операционной системы, а также рассматриваются два основных механизма управления доступом процессов к файлам.

1.1. Понятие операционной системы

Операционная система (operating system) – совокупность *системных* программ, предназначенная для обеспечения определенного уровня *эффективности* системы обработки информации за счет автоматизированного *управления* ее работой и предоставляемого пользователю определенного набора *услуг* [2].

Система обработки информации – совокупность технических средств и программного обеспечения, а также методов обработки информации и действий персонала, обеспечивающая выполнение автоматизированной обработки информации [2].

Из этих определений можно выделить следующие ***свойства операционной системы***:

1. Является основой построения системы автоматизированной обработки информации, называемой также *автоматизированной системой* или *информационной системой*.

2. Представляет собой набор программ, относящихся к классу *системного* программного обеспечения, и вместе с техническими средствами формирует вычислительную/компьютерную систему (computer system).

3. Управляет техническими средствами вычислительной системы, также называемыми *аппаратными средствами* или *платформами* (hardware), которые включают центральный процессор (central processing unit, CPU), оперативную память (random-access memory, RAM), устройства ввода/вывода и устройства хранения.

4. Предоставляет пользователю набор услуг, также называемых *сервисами*. Пользователь обращается к сервисам операционной системы для выполнения своих задач и запускает программы в среде операционной системы.

От того, насколько эффективно операционная система управляет аппаратными средствами и предоставляет сервисы пользователю, зависит эффективность системы в целом. Так как существует множество разнообразных задач обработки информации, невозможно с помощью одной операционной системы обеспечить приемлемую эффективность их выполнения во всех случаях. Если за точку отсчета истории операционных систем взять создание в компании General Motors операционной системы для мейнфрейма IBM 704 в 1956 г. [4], то за более чем шестидесятилетнюю историю разработано множество самых разных операционных систем, предназначенных для функционирования на разных аппаратных платформах и выполнения разных задач обработки информации.

Операционная система, которая позволяет одновременно и независимо выполнять несколько программ, принадлежащих двум или более пользователям, называется *многопользовательской* (multiuser) [11]. В ряде источников [11–14, 20] подробно рассматриваются функции, которыми должна обладать современная многопользовательская операционная система. К основным функциям можно отнести обеспечение:

- загрузки пользовательских программ в оперативную память и их исполнения;
- управления памятью;
- работы с устройствами долговременной памяти;
- более или менее стандартизованного доступа к различным периферийным устройствам;
- некоторого пользовательского интерфейса;
- механизма аутентификации для проверки личности пользователя;
- механизма управления доступом пользователей к ресурсам друг друга;
- механизма защиты от ошибочных пользовательских программ, которые могут блокировать другие программы, запущенные в системе;

- механизма защиты от вредоносных пользовательских программ, которые могут шпионить за деятельностью других пользователей или мешать им работать;
- механизма учета, ограничивающего количество ресурсов, выделенных каждому пользователю;
- механизма регистрации действий пользователей.

В данном пособии рассматривается многопользовательская операционная система общего назначения GNU/Linux, которая относится к POSIX-совместимым операционным системам, также называемым семейством UNIX.

1.2. Ядро операционной системы

Основная программа операционной системы называется *ядром* (kernel). Из всех программ операционной системы ядро самым первым загружается в RAM во время начальной загрузки системы (system boots) и содержит базовую функциональность для «обеспечения определенного уровня эффективности» системы в целом. Поэтому часто, говоря об операционной системе, подразумевают ее ядро, и наоборот. В нашем случае программа Linux является ядром операционной системы, а другие программы разрабатываются в рамках проекта GNU. Поэтому полным названием рассматриваемой операционной системы является GNU/Linux.

Ядро операционной системы выполняет две основные задачи:

1. Управление аппаратными средствами.
2. Предоставление среды выполнения для пользовательских программ, которые работают в компьютерной системе (создание иллюзии идеального компьютера).

В современных операционных системах пользовательская программа не может напрямую взаимодействовать с аппаратными средствами. Когда программа хочет обратиться к аппаратному ресурсу, она должна выполнить запрос к ядру операционной системы. Ядро выполнит различные проверки такого запроса и, в случае его легитимности, будет взаимодействовать с соответствующими аппаратными средствами от имени пользовательской программы [11].

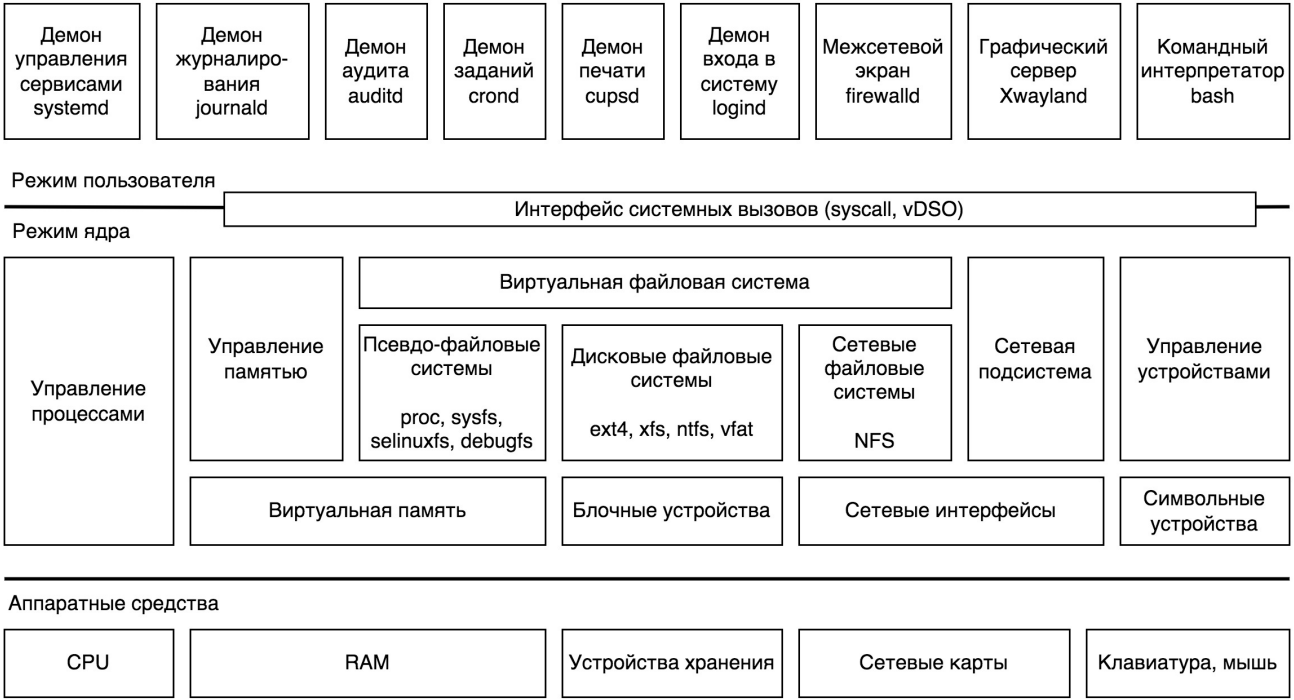


Рис. 1.1. Архитектура операционной системы GNU/Linux

Чтобы реализовать такой механизм, современные операционные системы полагаются на наличие определенных аппаратных функций, которые запрещают пользовательским программам напрямую взаимодействовать с низкоуровневыми компонентами оборудования или получать доступ к произвольным участкам RAM. В частности, аппаратное обеспечение предоставляет по крайней мере два различных режима выполнения для CPU: непривилегированный режим для пользовательских программ и привилегированный режим для ядра [11]. Эти режимы называются «*режим пользователя*» (user mode) и «*режим ядра*» (kernel mode) соответственно.

Таким образом, чтобы гарантировать невозможность обхода механизмов защиты, операционная система должна опираться на аппаратные механизмы защиты, предоставляемые CPU.

На рис. 1.1 представлена архитектура операционной системы GNU/Linux. Показаны основные подсистемы ядра, а также соответствующие им аппаратные средства. В дистрибутив GNU/Linux могут быть включены самые разнообразные программы. Иногда трудно провести четкую грань между программами операционной системы и программами пользователя. Все программы выполняются в виде процессов, и, с точки зрения управления процессами, системные и пользовательские программы ничем не отличаются. Тем не менее, на рис. 1.1 показаны основные программы операционной системы, без которых работа пользователей в системе была бы невозможна.

Обратите внимание на то, что процессы выполняются в непривилегированном режиме пользователя (в кольце 3), а ядро – в привилегированном режиме ядра (в кольце 0). Для взаимодействия процессов с ядром используется интерфейс системных вызовов.

1.3. Защищенный режим работы процессора x86-64

Защищенный режим процессора x86-64 позволяет защитить данные операционной системы от выполняющихся в ней процессов, а также защитить данные процессов друг от друга. В [15] рассматривается механизм разделения программ на разные уровни привилегий.

При работе в защищенном режиме CPU следит за правильным выполнением текущей программой ряда условий: например, она не

должна выполнять некоторые инструкции или обращаться к некоторым областям памяти. Если все же происходит нарушение какого-либо условия, то CPU генерирует специальный тип прерывания – так называемое исключение, и снабжает это прерывание информацией, описывающей, где произошло нарушение и как оно произошло. Затем специальная процедура (обработчик прерывания) обрабатывает это прерывание и решает, что дальше делать с программой (например, прекратить ее выполнение).

Определением условий работы и ограничений для программ должна заниматься операционная система. Когда программа переводит CPU в защищенный режим, то становится полноправным хозяином компьютера – по сути, операционной системой. Она устанавливает систему разрешений и условий для других программ. Для того чтобы эти условия и разрешения не смогла переопределить другая программа, в процессоре введена система уровней привилегий.

Благодаря тому, что в защищенном режиме можно задействовать механизм многозадачности, на CPU может выполняться одновременно несколько задач. Поэтому помимо самой операционной системы на CPU может выполняться еще несколько задач, и каждая может нарушить целостность данных и подвергнуть данные пользователя опасности; именно поэтому и введена система привилегий. Благодаря системе привилегий прикладная программа не может изменить правила, которые установила операционная система, и тем самым получить доступ к важным данным.

Основой защищенного режима являются *уровни привилегий* – степени использования ресурсов CPU. Всего таких уровней четыре, они имеют номера от 0 до 3. Уровень 0 – самый привилегированный. Когда программа работает на этом уровне привилегий, ей «можно все». Уровень 1 – менее привилегированный, и запреты, установленные на уровне 0, действуют для уровня 1. Уровень 2 – еще менее привилегированный, а 3-й имеет самый низкий приоритет [15].

В документации компании Intel уровни привилегий интерпретируются в виде концентрических колец защиты, где самое внутреннее кольцо соответствует нулевому уровню (т.е. самому привилегированному), а внешнее кольцо – третьему уровню [16].

Сегмент памяти описывается сегментным дескриптором (специальная структура размером 8 байт). В поле DPL (Descriptor Privilege Level) содержится уровень привилегий сегмента. Это 2-битовое

поле, в которое при создании дескриптора записывают значения от 0 до 3, определяющие уровень привилегий. Например, у кода, который выполняется на нулевом уровне привилегий, в дескрипторе сегмента кода оба бита должны быть сброшены. Именно по этим битам (и по уровню привилегий сегмента стека) процессор и определяет уровень привилегий текущего выполняемого кода [15].

Ядро Linux использует только два уровня привилегий: на нулевом выполняется код ядра, а на третьем – программы пользователей в контексте процессов.

1.4. Системные вызовы

Прикладным программным интерфейсом (application programming interface, API) ядра является интерфейс *системных вызовов*. С помощью системных вызовов ядро предоставляет программам пользователей (процессам) доступ к своим сервисам. Таким образом, системный вызов – это точка входа в ядро для выполнения функций ядра по запросу процесса.

Рассмотрим общие характеристики системных вызовов, приведенные в [9]:

- системный вызов изменяет состояние CPU, переводя его из пользовательского режима в режим ядра, позволяя таким образом CPU получить доступ к защищенной памяти ядра;
- набор системных вызовов жестко зафиксирован. Каждый системный вызов идентифицируется по уникальному номеру. Пользовательские программы обращаются к вызовам по их именам;
- у системного вызова может быть набор аргументов, определяющих информацию, которая должна быть передана из пространства пользователя (т.е. из виртуального адресного пространства процесса) в пространство ядра, и наоборот.

Существует несколько способов для перевода CPU из режима пользователя в режим ядра. В процессорах Intel исторически для этого использовалось прерывание int $0x80. Затем появилась пара инструкций SYSENTER и SYSEXIT. В современных процессорах x86-64 используются инструкции SYSCALL для передачи управления с третьего уровня на нулевой и SYSRET для быстрого возврата [16].

Обычно программы обращаются к системным вызовам через функции-обертки, которые предоставляет библиотека **libc**. А функции-обертки, в свою очередь, выполняют подготовительную работу и производят сам системный вызов. Если для какого-то системного вызова функция-обертка отсутствует, то обращение осуществляется через универсальную функцию **syscall()**:

```
#include <sys/syscall.h> /* Имена системных вызовов */
#include <unistd.h>

long syscall(long number, ...);
```

Функция **syscall()** выполняет системный вызов, обращаясь к функции ядра по ее порядковому номеру, передаваемому через аргумент **number**. За номером следуют аргументы конкретного системного вызова. Функция **syscall()** сохраняет регистры CPU перед выполнением системного вызова, восстанавливает регистры по возвращению из системного вызова, в случае успешного выполнения системного вызова возвращает 0, а в случае ошибки -1, и номер ошибки сохраняет в переменной **errno**.

На платформе x86-64 для передачи аргументов системным вызовам и получения кода возврата используются регистры CPU, представленные в табл. 1.1.

Таблица 1.1

Регистры для системного вызова на платформе x86-64

Номер вызова	arg1	arg2	arg3	arg4	arg5	arg6	Код возврата	Дополнительный код возврата
rax	rdi	rsi	rdx	r10	r8	r9	rax	rdx

Для указания системных вызовов по их именам нужно подключить заголовочный файл **<sys/syscall.h>**, который в зависимости от архитектуры подключит другие необходимые файлы. На платформе x86-64 номера системных вызовов перечислены в файле **/usr/include/asm/unistd_64.h**.

Ядро Linux предоставляет более 400 системных вызовов. Конкретное количество зависит от архитектуры (на платформе x86-64 их 442). В прил. 1 приведены номера системных вызовов, рассматриваемых в данном пособии, для платформы x86-64.

Чтобы проиллюстрировать применение инструкции **syscall**, рассмотрим пример классической программы, которая выводит строку «Hello, world!» на экран с помощью системного вызова **write()** и завершает свою работу с помощью системного вызова **exit()** с кодом возврата 0.

Листинг 1.1. Пример программы на ассемблере.

```
.data
msg:
    .ascii "Hello, world!\n"
/* Вычисляем длину строки */
    len = . - msg

.text
    .global _start

_start:
/* Заносим номер системного вызова write() в регистр
rax */
    movq  $1, %rax
/* Выводим в поток 1 (stdout), поэтому заносим 1 в
регистр rdi */
    movq  $1, %rdi
/* Передаем адрес строки в сегменте данных */
    movq  $msg, %rsi
/*  Передаем количество выводимых символов (длину
строки) */
    movq  $len, %rdx
/* Вызываем функцию ядра */
    syscall

/* Заносим номер системного вызова exit() в регистр
rax */
    movq  $60, %rax
/* Код возврата равен нулю (операция XOR) */
```

```
    xorq  %rdi, %rdi
/* Вызываем функцию ядра */
    syscall
```

Программа написана на ассемблере GNU, синтаксис которого отличается от синтаксиса ассемблера Intel. В программе определены две секции (два сегмента): данных и кода (текста). Сначала в регистры загружаются номер и аргументы для системного вызова **write()**, и выполняется инструкция **syscall**. Затем в регистры загружаются номер и единственный аргумент для системного вызова **exit()**, и также выполняется инструкция **syscall**. Скомпилируем и запустим программу:

```
$ as p_hello.S -o p_hello.o
$ ld p_hello.o -o p_hello
$ ./p_hello
Hello, world!
$ echo $?
0
```

Команда **as** вызывает GNU-ассемблер, а команда **ld** вызывает GNU-линковщик. В переменной BASH $? сохраняется код возврата завершившегося процесса. Видно, что программа вывела строку и завершилась с кодом возврата равным 0.

Несмотря на то, что для осуществления системного вызова на смену прерываниям пришли специализированные инструкции, переключение CPU из пользовательского режима в режим ядра является дорогостоящей операцией. Некоторые системные вызовы выполняются программами пользователя так часто, что начинают влиять на общую производительность системы. Для решения этой проблемы используется механизм виртуального динамического разделяемого объекта (**vDSO** – virtual dynamic shared object). **vDSO** представляет собой небольшую разделяемую библиотеку, которую ядро автоматически отображает в адресное пространство процесса, что позволяет процессу обращаться к функции ядра, не выполняя системного вызова.

Стандартная библиотека **glibc** скрывает от разработчика, какой метод используется для осуществления системного вызова. На платформе x86-64 увидеть **vDSO** можно в выводе команды **ldd** в виде разделяемой библиотеки **linux-vdso.so.1**.

На платформе x86-64 через механизм **vDSO** реализованы четыре системных вызова: **time()**, **clock_gettime()**, **gettimeofday()**, **getcpu()**. Небольшое количество объясняется тем, что эти вызовы с точки зрения безопасности не представляют потенциальных уязвимостей, так как они только получают информацию, которая доступна всем, и не изменяют состояние ядра.

Рассмотрим пример программы **vdso/gettimeofday.c** [9]. Программа позволяет сравнить производительность двух методов обращения к системному вызову **gettimeofday()**. По умолчанию библиотека **glibc** обращается к механизму **vDSO**, но с помощью опций компилятора можно собрать программу, в которой используется универсальная функция-обертка **syscall()** для осуществления системного вызова **gettimeofday()**.

Листинг 1.2. Программа vdso/gettimeofday.c.

```
int main(int argc, char *argv[])
{
    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s loop-count\n", argv[0]);

    long lim = atoi(argv[1]);

    for (int j = 0; j < lim; j++) {
        struct timeval curr;

#ifdef USE_SYSCALL
        if (syscall(__NR_gettimeofday, &curr, NULL) ==
-1)
            errExit("gettimeofday");
#else
        if (gettimeofday(&curr, NULL) == -1)
            errExit("gettimeofday");
#endif
    }

    exit(EXIT_SUCCESS);
}
```

Запустим обе программы. Для наглядности выполним системный вызов **gettimeofday()** 100 млн раз. С помощью команды **time** получим общее время выполнения программ:

```
$ time ./syscall_gettimeofday 100000000

real  0m17.659s
user  0m2.684s
sys   0m14.956s
$ time ./vdso_gettimeofday    100000000

real  0m1.591s
user  0m1.587s
sys   0m0.002s
```

Видно, что при выполнении системного вызова **gettimeofday()** стандартным способом потребовалось 17 с, а с помощью механизма **vDSO** – меньше двух секунд.

1.5. Понятие защищенной операционной системы

Для определения понятия защищенной операционной системы необходимо учитывать взаимное влияние механизмов защиты, которыми она обладает, и угроз информационной безопасности, которым она противостоит, так как «безопасность не является абсолютной характеристикой и может рассматриваться только относительно некоторой среды, в которой действуют определенные угрозы» [5].

В [5] формулируются три свойства, которыми должна обладать защищенная система обработки информации. Такая система:

- осуществляет автоматизацию некоторого процесса обработки конфиденциальной информации, включая все аспекты этого процесса, связанные с обеспечением безопасности обрабатываемой информации;
- успешно противостоит угрозам безопасности, действующим в определенной среде;
- соответствует требованиям и критериям стандартов информационной безопасности.

Как было показано, операционная система является основой системы обработки информации, поэтому исходя из свойств защищенной системы обработки информации, будем называть *защищенной операционной системой* такую операционную систему, которая обладает средствами защиты информации, способными противостоять заданным угрозам безопасности, а также соответствует требованиям стандартов информационной безопасности.

Первым стандартом информационной безопасности стали «Критерии безопасности компьютерных систем» (Trusted Computer System Evaluation Criteria, TCSEC) [8], также известные по цвету обложки как «Оранжевая книга», разработанные в США в 1983 г. В России в 1992 г. были выпущены документы под общим названием «Руководящие документы Государственной технической комиссии» (РД ГТК). Подробный анализ стандартов информационной безопасности представлен в [5], [6].

В защищенной операционной системе должна быть определена *политика безопасности*. В [5] под *политикой безопасности* понимается совокупность норм и правил, обеспечивающих эффективную защиту системы обработки информации от заданного множества угроз безопасности. Для формального выражения политики безопасности используют модели безопасности. Можно выделить два основных класса моделей: дискреционные и мандатные. Подробный анализ моделей безопасности представлен в [5], [7].

Важным понятием защищенной системы обработки информации является ядро безопасности (Trusted Computing Base, TCB), которое в [8] называют «сердцем доверенной компьютерной системы». В соответствии с [8], TCB содержит все элементы системы, ответственные за реализацию политики безопасности и поддержку изоляции объектов (кода и данных), на которых основана защита. Границы TCB определяют «периметр безопасности» компьютерной системы. TCB должно быть как можно проще в соответствии с функциями, которые оно должно выполнять. Таким образом, TCB включает в себя аппаратное (hardware), микропрограммное (firmware) и программное (software) обеспечение, критически важные для защиты.

При сертификации операционной системы на соответствие требованиям стандартов информационной безопасности состав TCB формирует объект оценки. Поэтому важно правильно определить

границы и состав TCB. TCB должно быть спроектировано и реализовано таким образом, чтобы при исключении из него системных элементов от этих элементов не требовалось бы быть доверенными для обеспечения защиты [8]. На рис. 1.2 показан состав TCB операционной системы GNU/Linux.

Одним из базовых принципов, лежащем в основе проектирования TCB, является принцип наименьших привилегий (principle of least privilege), в соответствии с которым каждому элементу системы предоставляются только минимально необходимые авторизации для его успешного функционирования. Применительно к пользователям этот принцип означает, что пользователь получает доступ только к тем ресурсам системы, которые минимально необходимы для успешного выполнения его задач.

Поверхность атаки (attack surface) – все потенциально уязвимые элементы системы, через которые злоумышленник может проникнуть в систему и извлечь информацию из системы. Одной из задач информационной безопасности является уменьшение поверхности атаки при сохранении нормального функционирования системы. Например, если в системе установлена программа, которая не используется, ее лучше удалить, так как рано или поздно в этой программе будут обнаружены ошибки или уязвимости. Если в системе подняты сервисы, которые не используются, их лучше отключить по той же причине, особенно если речь идет про сетевые сервисы.

С точки зрения информационной безопасности все сущности системы можно разделить на *субъекты* и *объекты доступа*. В операционной системе GNU/Linux в качестве субъекта выступает процесс, который действует от имени определенного пользователя, запустившего на выполнение некоторую программу. Под *объектом* понимается некоторый системный ресурс, чаще всего файл, который принадлежит определенному пользователю. В отдельных случаях объектом доступа может выступать другой процесс, когда по отношению к нему выполняют какие-то действия другие процессы (например, отправка сигнала). TCB сравнивает атрибуты безопасности процесса и файла и запрашиваемый вид доступа и на основе политики безопасности разрешает или запрещает доступ.

Рис. 1.2. TCB операционной системы GNU/Linux

В табл. 1.2 представлено сравнение дискреционного и мандатного управления доступом.

Таблица 1.2

Сравнение дискреционного и мандатного управления доступом

Дискреционное управление доступом	**Мандатное управление доступом**
Процесс выполняется от имени определенного пользователя. У файла (ресурса) есть владелец (обычно пользователь, который создал файл)	Процессу и файлу (ресурсу) присваивается контекст безопасности (метка)
Владелец по своему усмотрению (discretion) устанавливает права доступа к файлу (ресурсу)	Доступ к файлу (ресурсу) определяется на основе сравнения контекстов безопасности (меток) файла (ресурса) и процесса. Алгоритм сравнения прописан в политике безопасности
Процессы взаимодействуют на основе сравнения идентификаторов пользователей	Процессы взаимодействуют на основе сравнения их контекстов безопасности (меток) и правил политики безопасности

1.6. Управление доступом

Так как управление доступом базируется на сравнении атрибутов безопасности процессов и файлов, важно понимать состав этих атрибутов, и при каких проверках на доступ используются те или иные атрибуты.

Атрибуты безопасности процесса и структуры данных ядра, в которых они хранятся, представлены в табл. 1.3.

На рис. 1.3 показаны структуры данных ядра, в которых хранятся атрибуты безопасности процесса, и связи между ними.

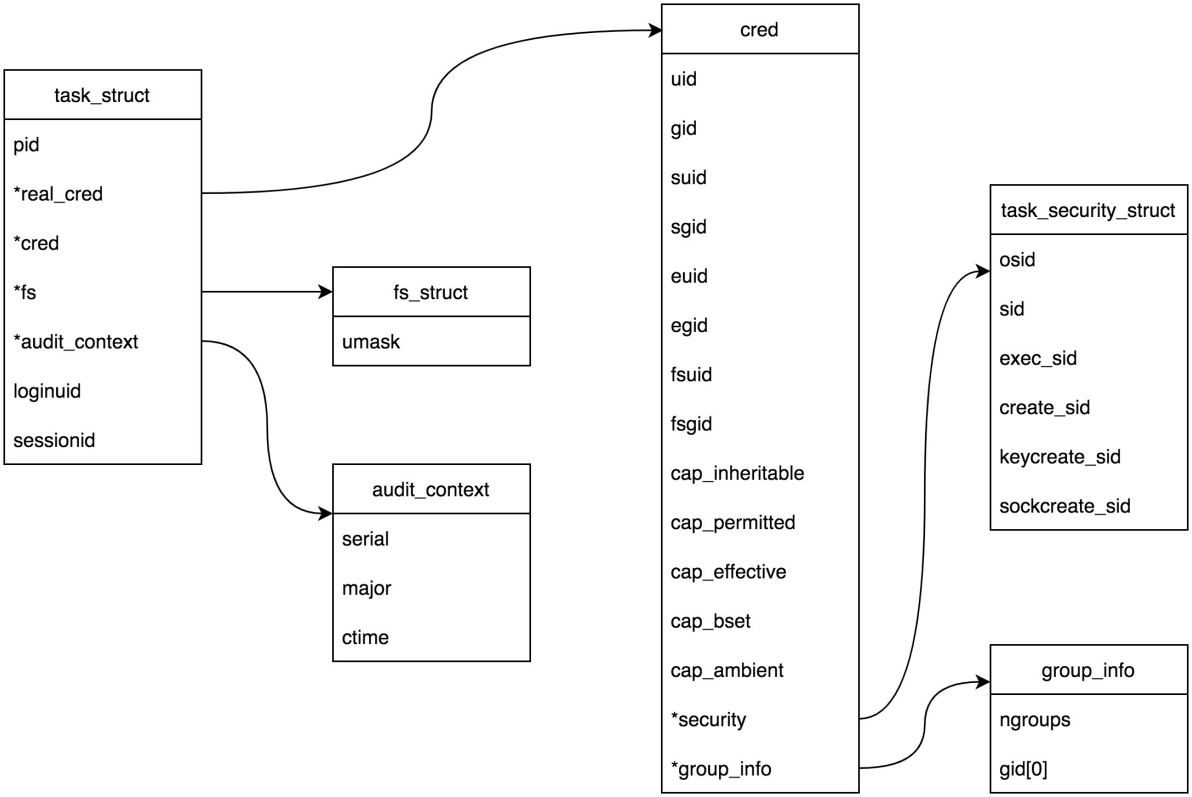


Рис. 1.3. Атрибуты безопасности процесса

Таблица 1.3

Атрибуты безопасности процесса

Атрибут	**Структура данных**	**Назначение**
uid	cred	Реальный идентификатор пользователя
gid	cred	Реальный идентификатор группы
suid	cred	Сохраненный идентификатор пользователя
sgid	cred	Сохраненный идентификатор группы
euid	cred	Эффективный идентификатор пользователя
egid	cred	Эффективный идентификатор группы
fsuid	cred	Идентификатор пользователя для доступа к файлам (VFS)
fsgid	cred	Идентификатор группы для доступа к файлам (VFS)
cap_inheritable	cred	Наследуемые привилегии
cap_permitted	cred	Разрешенные привилегии (которые процесс может использовать)
cap_effective	cred	Эффективные привилегии (которые процесс фактически использует в данный момент)
cap_bset	cred	Ограничивающие привилегии
cap_ambient	cred	Привилегии внешней среды
*security	cred	Дополнительные атрибуты из модулей LSM
group_info	group_info	Дополнительные группы процесса

Окончание табл. 1.3

Атрибут	Структура данных	Назначение
umask	fs_struct	Пользовательская маска создания файлов процесса
*audit_context	task_struct	Контекст аудита
loginuid	task_struct	Идентификатор, с которым пользователь выполнил вход в систему. Используется системой аудита
sessionid	task_struct	Идентификатор сессии пользователя. Используется системой аудита

Свойства файла определяет файловая система, в которой он хранится. Для файловой системы **ext4**, являющейся стандартом de facto в дистрибутивах GNU/Linux, атрибуты безопасности файла представлены в табл. 1.4.

Таблица 1.4

Атрибуты безопасности файла в файловой системе **ext4**

Атрибут	Структура данных	Назначение
i_uid	ext4_inode	Идентификатор владельца файла
i_gid	ext4_inode	Идентификатор группы – владельца файла
i_mode	ext4_inode	Биты доступа
i_file_acl_*	ext4_inode	ACL, привилегии и контекст безопасности SELinux
i_flags	ext4_inode	Флаги файла

В табл. 1.5 показаны особенности принятия решения о доступе в DAC и SELinux. Обратите внимание на то, что процесс сможет получить доступ к файлу, только если оба механизма DAC и SELinux разрешат доступ.

Таблица 1.5

Сравнение механизмов управления доступом DAC и SELinux

Аспект	DAC	SELinux
Атрибуты безопасности процесса	UID, EUID, FSUID, GID, EGID, FSGID, дополнительные группы, привилегии	Контекст безопасности (домен) процесса
Атрибуты безопасности файла	UID, GID, биты доступа, ACL	Контекст безопасности (тип) файла
Решение о доступе	Сравнение идентификаторов FSUID, FSGID и дополнительных групп процесса с идентификаторами UID, GID, битов доступа и ACL файла. Анализ привилегий процесса	Сравнение контекстов безопасности процесса и файла

Контрольные вопросы

1. Чем отличается защищенная операционная система от незащищенной?
2. Для чего предназначено ядро операционной системы?
3. На каком уровне привилегий выполняется ядро операционной системы?
4. Каким образом процессы взаимодействуют с ядром операционной системы?
5. Как формируется политика безопасности?
6. Назовите основные классы моделей безопасности.

7. Какие атрибуты процесса используются при управлении доступом?

8. Чем характеризуется мандатное управление доступом?

9. Какие элементы системы должны быть включены в ТСВ?

10. Какими правами должен обладать пользователь в соответствии с принципом наименьших привилегий? Приведите пример.

Глава 2. Управление доступом к файлам

В данной главе рассматривается понятие файла применительно к файловой системе **ext4**, и описываются механизмы управления доступом к файлам, представляющие важное значение с точки зрения безопасности операционной системы.

2.1. Индексный дескриптор файла

Основной структурой данных, описывающей файл, является структура **ext4_inode**, называемая *индексным дескриптором* (от index node, inode), или просто *инодом*.

Иноды хранятся в файловой системе в виде линейного массива, называемого *таблицей инодов*. Поиск (адресация) файла в рамках файловой системы осуществляется по номеру **inode** (т.е. его смещению) в таблице **inode**. Нумерация начинается с единицы. Существует несколько зарезервированных номеров инодов, представленных в табл. 2.1.

Таблица 2.1

Зарезервированные номера индексных дескрипторов

Константа	Номер	Описание
EXT4_BAD_INO	1	Плохие блоки
EXT4_ROOT_INO	2	Инод корневой директории файловой системы
EXT4_USR_QUOTA_INO	3	Не используется
EXT4_GRP_QUOTA_INO	4	Не используется
EXT4_BOOT_LOADER_INO	5	Начальный загрузчик
EXT4_UNDEL_DIR_INO	6	Не используется
EXT4_RESIZE_INO	7	Зарезервированный дескриптор группы
EXT4_JOURNAL_INO	8	Журнал

Ядро Linux поддерживает семь типов файлов, представленных в табл. 2.2.

Таблица 2.2

Типы файлов

Название	Значение в **ext4.h**	Константа в **stat.st_mode**	Значение в **stat.st_mode**
Обычный файл	1	S_IFREG	0100000
Директория	2	S_IFDIR	0040000
Файл символьного устройства	3	S_IFCHR	0020000
Файл блочного устройства	4	S_IFBLK	0060000
Именованный канал (также называемый FIFO)	5	S_IFIFO	0010000
UNIX-сокет	6	S_IFSOCK	0140000
Символьная ссылка	7	S_IFLNK	0120000

Существует ряд макросов для определения типа файла. Например, системный вызов **stat()** возвращает в поле **stat.st_mode** тип файла и его права доступа. Чтобы проверить, является ли файл обычным файлом, можно написать в программе:

```
stat(pathname, &sb);
if (S_ISREG(sb.st_mode)) {
/* Работа с обычным файлом */
}
```

Файлы всех типов описываются одной и той же структурой **ext4_inode**. Однако, в зависимости от конкретного типа файла, некоторые поля этой структуры интерпретируются по-разному. Рассмотрим основные поля структуры **ext4_inode** в порядке вывода команды **ls -l** (в длинном формате).

Пример:

```
$ ls -l /etc/passwd
-rw-r--r--. 1 root root 1908 Jun 30 06:00 /etc/passwd
```

В табл. 2.3 показаны основные поля структуры **ext4_inode**. Для вывода номера инода файла предназначена команда **ls -i**. Как уже было сказано, сам номер инода в иноде не содержится – это индекс в массиве инодов.

Таблица 2.3

Основные поля структуры **ext4_inode**

Название	Назначение
i_mode	Тип файла и права доступа
i_links_count	Количество жестких ссылок на файл. Максимальное количество ссылок на файл – 65000
i_uid l_i_uid_high	Идентификатор владельца файла
i_gid l_i_gid_high	Идентификатор группы – владельца файла
i_size_lo i_size_high	Размер файла в байтах
i_atime i_atime_extra	Время доступа к файлу (чтения файла)
i_ctime i_ctime_extra	Время модификации инода (метаданных файла)
i_mtime i_mtime_extra	Время модификации файла (записи в файл)
i_dtime	Время удаления файла
i_blocks_lo l_i_blocks_high	Количество блоков
i_flags	Флаги файла

Окончание табл. 2.3

Название	Назначение
i_block[EXT4_N_BLOCKS]	Номера блоков с содержимым файла (EXT4_N_BLOCKS=15)
i_file_acl_lo l_i_file_acl_high	Ссылка на расширенные атрибуты файла
i_extra_isize	Размер инода сверх 128 байт
i_crtime i_crtime_extra	Время создания файла

В файловых системах **ext2** и **ext3** размер инода был фиксированным и составлял 128 байт. Для его хранения на диске выделялась запись такого же размера.

В файловой системе **ext4** для хранения инода выделяется запись, размер которой превышает размер структуры **ext4_inode**. Размер записи хранится в поле **s_inode_size** структуры **ext4_super_block** (суперблока файловой системы).

Количество байтов, фактически используемых структурой **ext4_inode** сверх изначальных 128 байт, хранится в поле **i_extra_isize** инода, что позволяет структуре **ext4_inode** расширяться в будущем без необходимости переформатирования файловой системы.

По умолчанию, в файловой системе **ext4** для хранения каждого инода выделяется запись размером 256 байт, а размер структуры **ext4_inode** равен 160 байт (i_extra_isize = 32). Свободное место между концом структуры **ext4_inode** и концом записи может использоваться для хранения расширенных атрибутов (256 – 160 = = 96 байт). Теоретически каждая запись может занимать размер целого блока файловой системы, хотя на практике это было бы неэффективно. На рис. 2.1 показаны основные поля структуры **ext4_inode**, а также ее размер и размер записи на диске для хранения инода.

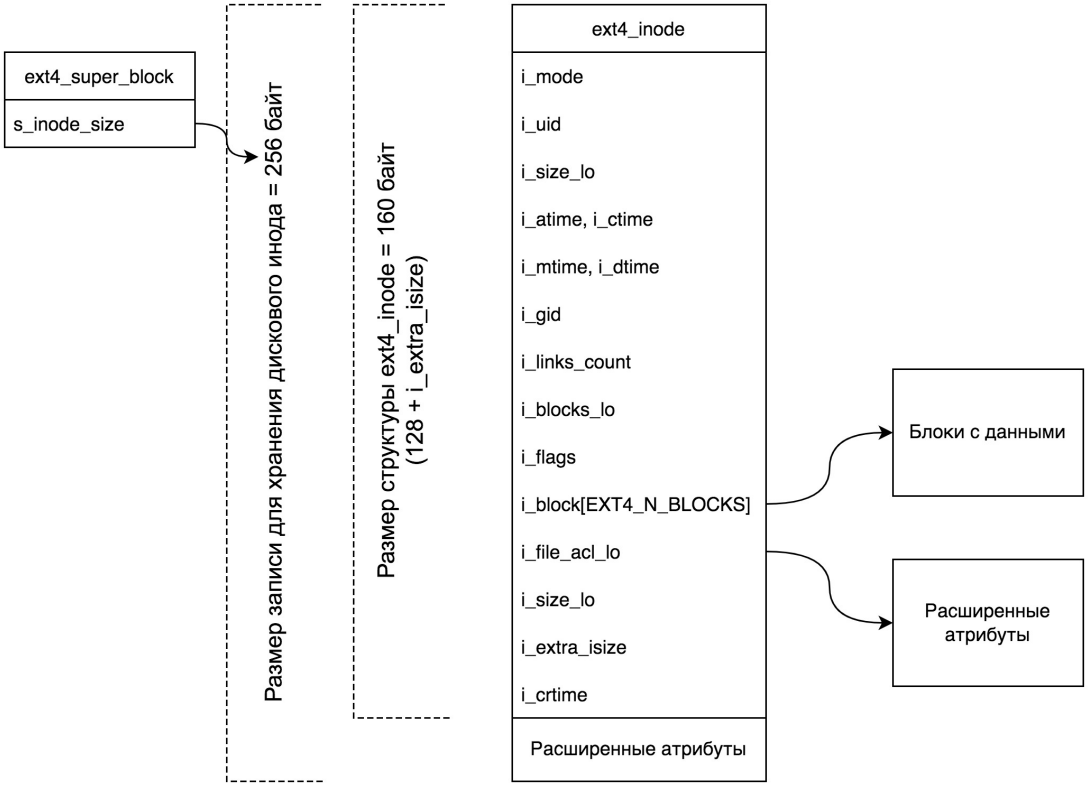


Рис. 2.1. Основные поля структуры **ext4_inode**

2.2. Владелец и группа-владелец файла

С точки зрения DAC каждый файл принадлежит одному определенному пользователю и одной определенной группе пользователей. Поэтому в иноде файла хранятся два идентификатора: идентификатор владельца (UID) и идентификатор группы-владельца (GID).

При создании нового файла (например, с помощью вызовов **open()** или **mkdir()**) его владелец (UID) устанавливается равным FSUID создающего процесса.

Группа-владелец (GID) нового файла зависит от типа файловой системы, опций монтирования и наличия бита **set-group-ID** у родительской директории.

Если файловая система **ext4** смонтирована с опцией **-o grpid**, то GID нового файла устанавливается равным GID родительской директории.

По умолчанию файловая система **ext4** монтируется с опцией -o **nogrpid**. В этом случае GID нового файла устанавливается равным FSGID создающего процесса, но если у родительской директории установлен бит **set-group-ID**, то GID нового файла устанавливается равным GID родительской директории.

Для изменения владельца и группы-владельца файла используется системный вызов **chown()**:

```
#include <unistd.h>

int chown(const char *pathname, uid_t owner, gid_t group);
```

Аргумент **pathname** задает путь к файлу. Аргумент **owner** задает новый UID, а аргумент **group** – новый GID. Если значение одного из идентификаторов равно –1, то этот идентификатор в иноде не меняется.

Только процесс с привилегией CAP_CHOWN может изменить владельца файла, а также изменить группу-владельца на произвольную другую группу. Владелец файла может изменить только группу-владельца и только на ту группу, в состав которой он входит сам.

Если у исполняемого файла меняются владелец или группа-владелец, то биты **set-user-ID** и **set-group-ID** сбрасываются. Это повышает безопасность и гарантирует, что обычный пользователь не сможет на свой собственный исполняемый файл установить биты **set-user-ID** или **set-group-ID**, а затем передать данный файл пользователю или группе **root** и тем самым повысить свои привилегии.

Если у исполняемого файла меняются владелец или группа-владелец, то все наборы привилегий очищаются.

В случае успеха возвращается 0. В случае ошибки возвращается -1, и в переменную **errno** записывается код ошибки.

Исторически в Linux системный вызов **chown()** поддерживал только 16-битные идентификаторы пользователя и группы. Позже был добавлен системный вызов **chown32()**, поддерживающий 32-битные идентификаторы. Библиотечная функция-обертка **chown()** скрывает от разработчика эти детали.

Рассмотрим пример программы, приведенной в справочной странице **chown(2)**. Эта программа у файла, передаваемого во втором аргументе, меняет значение владельца на значение, передаваемое в первом аргументе. Напомним, что **argv[0]** – это имя программы.

Листинг 2.1. Изменение владельца файла.

```
#include <pwd.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int
main(int argc, char *argv[])
{
    uid_t uid;
    struct passwd *pwd;
    char *endptr;

/* Проверить наличие двух аргументов */
    if (argc != 3 || argv[1][0] == '\0') {
        fprintf(stderr,    "%s    <owner>    <file>\n",
argv[0]);
        exit(EXIT_FAILURE);
    }
```

```
/* Конвертировать первый аргумент из строки в длинное
целое */
    uid = strtol(argv[1], &endptr, 10);

/* Если в первом аргументе оказался не числовой иден-
тификатор владельца, а его имя, получить UID по имени
пользователя в файле /etc/passwd */
    if (*endptr != '\0') {
/* Получить UID по имени пользователя */
        pwd = getpwnam(argv[1]);
        if (pwd == NULL) {
            perror("getpwnam");
            exit(EXIT_FAILURE);
        }
        uid = pwd->pw_uid;
    }

/* Изменить владельца файла. Путь к файлу берется из
второго аргумента. -1 означает, что изменять группу-
владельца не нужно */
    if (chown(argv[2], uid, -1) == -1) {
        perror("chown");
        exit(EXIT_FAILURE);
    }

    exit(EXIT_SUCCESS);
}
```

Для изменения владельца и/или группы-владельца файла предназначена команда **chown**.

`chown [опции] [владелец][:[группа]]` список_файлов
`chown [опции] --reference=файл-образец` список_файлов

В одной команде можно указать новых владельца и группу-владельца через двоеточие, а можно – только владельца или только группу-владельца. Владелец и группа-владелец могут быть заданы как своими именами, так и числовыми значениями. Различные сочетания и результаты выполнения команды представлены в табл. 2.4.

Таблица 2.4

Формат команды **chown**

Формат	Результат
владелец	Меняется владелец каждого файла. Группа-владелец не меняется
владелец:группа	Меняется владелец и группа-владелец каждого файла
владелец:	Меняется владелец каждого файла. Группа-владелец каждого файла меняется на основную группу нового владельца
:группа	Владелец не меняется. Группа-владелец каждого файла меняется на новую группу
: или аргумент опущен	Ничего не меняется

С опцией **--reference** меняются владелец и группа-владелец на владельца и группу файла-образца. С опцией **-R** (в длинном виде **--recursive**) команда выполняется рекурсивно.

Для изменения группы-владельца файла предназначена отдельная команда **chgrp**:

```
chgrp [опции] группа список_файлов
chgrp [опции] --reference=файл-образец список_файлов
```

У каждого файла меняется группа-владелец на новое значение. С опцией **-R** (в длинном виде **--recursive**) команда выполняется рекурсивно. С опцией **--reference** меняется группа-владелец на группу файла-образца.

2.3. Биты прав доступа файла

С точки зрения DAC с помощью битов прав доступа можно разграничить доступ к файлу для трех категорий пользователей:

- владелец (owner или user) файла;
- группа-владелец (group) файла (т.е. пользователи, входящие в эту группу);
- все остальные пользователи (other).

Для каждой категории пользователей задаются три права доступа к файлу:

- чтение (read);
- запись (write);
- исполнение (execute) для обычных файлов и поиск (search) для директорий.

Кроме того, файлу назначаются три специальных права:

- бит **set-user-ID** (S_ISUID, 04000);
- бит **set-group-ID** (S_ISGID, 02000);
- **sticky-бит** (S_ISVTX, 01000).

Таким образом, для каждого файла формируется вектор из 12 прав доступа, которые хранятся в младших 12 битах поля **i_mode** инода файла. Если значение бита равно 1, то право предоставлено, а если 0, то право не предоставлено. Для обычного файла и директории некоторые права интерпретируются по-разному. В табл. 2.5 показаны название и назначение битов прав доступа.

Таблица 2.5

Биты прав доступа

Биты	Обычный файл	Директория
r (read)	Читать из файла	Выводить содержимое директории (только имена файлов)
w (write)	Писать в файл	Создавать, удалять, перемещать и переименовывать файлы в директории. Также требуется право на поиск

Окончание табл. 2.5

Биты	Обычный файл	Директория
x (exec)	Запускать файл на исполнение (файл является программой или сценарием)	Читать метаданные файлов в директории (содержимое инодов). Использовать директорию в пути к файлу, переходить в директорию (cd), выводить содержимое директории в длинном формате (ls -l)
u+s (set-user-ID)	Устанавливает эффективный идентификатор пользователя EUID процесса при исполнении файла равным идентификатору владельца файла UID	Не используется
g+s (set-group-ID)	Устанавливает эффективный идентификатор группы EGID процесса при исполнении файла равным идентификатору группы файла GID	Устанавливает идентификатор группы GID для файлов, создаваемых в директории, равным идентификатору группы директории GID (при создании файла, GID наследуется от директории, в которой создается файл)
o+t (sticky)	Не используется	Удалять или переименовывать файлы в директории может только владелец директории, или владелец файла, или процесс с привилегией CAP_FOWNER. Также требуется право на запись

Чтобы запустить двоичный файл, достаточно только права на исполнение, а для запуска сценария необходимо наличие двух прав: на чтение и на исполнение.

При доступе к файлу разрешение на поиск необходимо для всех директорий, которые содержатся в пути к файлу.

Право доступа на чтение директории позволяет только вывести список имен файлов. Чтобы получить доступ к самим файлам (т.е. к содержимому их инодов), нужно право на поиск в директории. И наоборот, при наличии права на поиск, но отсутствии права на чтение, можно получить доступ к файлу в директории, если известно его имя, однако нельзя вывести содержимое данной директории. Такой способ используется для управления доступом к содержимому общедоступных директорий.

Установка бита **set-group-ID** на директорию используется для организации директорий, предназначенных для работы над совместным проектом нескольких пользователей, включенных в одну общую группу.

Установка **sticky-**бита используется для организации общедоступных директорий, таких как **/tmp**. В таких директориях отдельный пользователь может создавать и удалять собственные файлы, но не может удалять файлы, принадлежащие другим пользователям. Имя константы S_ISVTX исторически происходит от **saved-text** (сохраненный текст).

Таким образом, простая схема битов прав доступа позволяет контролировать следующие виды доступа процесса к файлу:

- создание, переименование, перемещение и удаление файла;
- чтение метаданных файла (содержимого инода);
- чтение содержимого файла;
- изменение содержимого файла;
- запуск файла на исполнение.

Права доступа к файлу в выводе большинства команд показаны в виде строки из 10 символов (-rwxrwxrwx). Первый символ показывает тип файла (‘-’ – обычный файл, ‘d’ – директория). Следующие три символа (rwx) показывают наличие прав для владельца, следующие три символа (rwx) – для группы и последние три символа – для остальных пользователей. Каждый из трех символов соответствует одному праву: ‘r’ – чтение, ‘w’ – запись, ‘x’ – исполнение (поиск для директории). Если право предоставлено, то выводится соответствующий символ, если право не предоставлено, то вместо символа выводится знак ‘-’.

Права доступа и принадлежность файла можно просмотреть с помощью команд **ls -l** или **stat**. Например:

```
$ ls -l /etc/passwd
-rw-r--r--. 1 root root 2699 Oct 30 04:23 /etc/passwd
$ stat /etc/passwd
  File: /etc/passwd
  Size: 2699            Blocks: 8          IO Block:
4096   regular file
Device: 10303h/66307d   Inode: 5506543     Links: 1
Access: (0644/-rw-r--r--)  Uid: (    0/    root)
Gid: (    0/    root)
Context: system_u:object_r:passwd_file_t:s0
Access: 2022-01-05 01:59:07.338651785 +0300
Modify: 2021-10-30 04:23:12.796595929 +0300
Change: 2021-10-30 04:23:12.797595922 +0300
 Birth: -
```

В заголовочном файле **<sys/stat.h>** (/usr/include/sys/stat.h) определены константы для битов прав доступа. В табл. 2.6 показаны константы, их восьмеричные значения и назначение битов прав доступа.

Таблица 2.6

Константы для битов прав доступа к файлу

Константа	Восьмеричное значение	Назначение
S_ISUID	04000	set-user-ID (установить процессу эффективный UID при запуске)
S_ISGID	02000	set-group-ID (установить процессу эффективный GID при запуске)
S_ISVTX	01000	sticky-бит (историческое название от выражения *save swapped text after use*)
S_IRWXU	00700	Владелец файла (user) имеет право на чтение, запись и исполнение

Окончание табл. 2.6

Константа	Восьмеричное значение	Назначение
S_IRUSR	00400	Владелец файла (user) имеет право на чтение
S_IWUSR	00200	Владелец файла (user) имеет право на запись
S_IXUSR	00100	Владелец файла (user) имеет право на исполнение (для файла) или поиск (для директории)
S_IRWXG	00070	Группа имеет право на чтение, запись и исполнение
S_IRGRP	00040	Группа имеет право на чтение
S_IWGRP	00020	Группа имеет право на запись
S_IXGRP	00010	Группа имеет право на исполнение (для файла) или поиск (для директории)
S_IRWXO	00007	Остальные (others) пользователи имеют право на чтение, запись и исполнение
S_IROTH	00004	Остальные (others) пользователи имеют право на чтение
S_IWOTH	00002	Остальные (others) пользователи имеют право на запись
S_IXOTH	00001	Остальные (others) пользователи имеют право на исполнение (для файла) или поиск (для директории)

Рассмотрим пример. Создадим новую директорию и один файл в ней. Затем будем менять права доступа к директории и пробовать выполнять разные действия над директорией и содержащимся в ней файлом.

Обратите внимание на то, что команда **ls** может быть алиасом на команду **ls -l**. Вывести алиасы можно с помощью команды **alias**, а запустить саму команду, даже если ее именем назван алиас, можно, указав перед командой символ ‘\’.

```
$ mkdir dir1
$ ls -ld dir1
drwxrwxr-x. 2 user1 user1 4096 Feb  5 04:13 dir1
$ touch dir1/file1
$ ls -l dir1
total 0
-rw-rw-r--. 1 user1 user1 0 Feb  5 04:13 file1
```

Установим только для владельца директории только право на поиск:

```
$ chmod 100 dir1
$ ls -ld dir1
d--x------. 2 user1 user1 4096 Feb  5 04:13 dir1
$ ls dir1
ls: cannot open directory 'dir1': Permission denied
$ ls -l dir1
ls: cannot open directory 'dir1': Permission denied
$ ls dir1/file1
dir1/file1
$ ls -l dir1/file1
-rw-rw-r--. 1 user1 user1 0 Feb  5 04:13 dir1/file1
$ touch dir1/file2
touch: cannot touch 'dir1/file2': Permission denied
$ rm dir1/file1
rm: cannot remove 'dir1/file1': Permission denied
$ mv dir1/file1 dir1/file11
mv: cannot move 'dir1/file1' to 'dir1/file11': Per-
mission denied
```

Видно, что можно получить доступ к самому файлу, но какие-либо действия в директории выполнять нельзя. Установим только для владельца директории только право на запись:

```
$ chmod 200 dir1
$ ls -ld dir1
d-w-------. 2 user1 user1 4096 Feb  5 04:13 dir1
$ touch dir1/file2
touch: cannot touch 'dir1/file2': Permission denied
$ rm dir1/file1
rm: cannot remove 'dir1/file1': Permission denied
$ mv dir1/file1 dir1/file11
mv: failed to access 'dir1/file11': Permission denied
$ chmod 300 dir1
$ ls -ld dir1
d-wx------. 2 user1 user1 4096 Feb  5 04:13 dir1
$ touch dir1/file2
$ mv dir1/file1 dir1/file11
$ ls -l dir1/
ls: cannot open directory 'dir1/': Permission denied
$ rm dir1/file11
```

Видно, что для создания или удаления файлов в директории необходимо иметь разрешение и на поиск, и на запись для данной директории. **Обратите внимание: не обязательно иметь какие-либо права доступа к самому файлу, чтобы удалить его!**

Установим только для владельца директории только право на чтение:

```
$ chmod 400 dir1
$ ls -ld dir1
dr--------. 2 user1 user1 4096 Feb  5 04:17 dir1
$ ls dir1/
file2
$ ls -l dir1/
ls: cannot access 'dir1/file2': Permission denied
total 0
-????????? ? ? ? ?            ? file2
```

Видно, что в этом случае можно только вывести имя файла, но нельзя получить доступ к его метаданным (его иноду).

Для поиска файлов используется команда **find**. Если нужно найти файлы с определенными правами доступа, нужно использовать выражение **-perm**. Существует три формы записи выражения **-perm**:

```
-perm права
```

Биты прав доступа файла точно совпадают с аргументом права:

```
-perm -права
```

Все из заданных битов прав доступа файла должны присутствовать у файла:

```
-perm /права
```

Любой из заданных битов прав доступа файла должен присутствовать у файла.

Права можно задавать в восьмеричном или символьном виде. Символьный вид по синтаксису аналогичен команде **chmod**.

Например, следующая команда найдет все файлы и директории с установленным битом **set-user-ID** и сохранит информацию о них в файле **/root/suid.txt** в формате «режим_доступа владелец путь_к_файлу»:

```
# find / -perm -4000 -fprintf /root/suid.txt
'%#m %u %p\n'
# cat /root/suid.txt
04755 root /usr/bin/passwd
04755 root /usr/bin/chage
04755 root /usr/bin/mount
04755 root /usr/bin/umount
...
```

Найти файлы в текущей директории, у которых установлены права на чтение и запись для владельца и группы-владельца, а остальные пользователи могут только читать:

```
$ find . -perm 664
```

2.4. Алгоритм проверки прав доступа

Алгоритм проверки прав при доступе процесса к файлу рассмотрен в [9]. Ядро проверяет права доступа к файлу каждый раз при указании его имени в системном вызове. Если путь к файлу содержит директории, то помимо проверки прав доступа к самому файлу ядро проверяет также право на поиск для каждой директории в путевом имени. Проверки прав доступа выполняются благодаря использованию файловых идентификаторов пользователя и группы, а также дополнительных групп процесса.

Как только файл открывается с помощью системного вызова **open()**, последующие системные вызовы (например, **read()**, **write()**, **mmap()**), которые работают с возвращенным дескриптором файла, не выполняют проверку прав доступа.

Алгоритм проверки прав доступа:

1. Если процесс является привилегированным, то предоставляется полный доступ.
2. Если FSUID для процесса совпадает с идентификатором владельца файла UID, то предоставляется доступ в соответствии с правами доступа для владельца файла.
3. Если FSGID или идентификатор любой дополнительной группы для процесса совпадает с идентификатором группы – владельца файла GID, то доступ предоставляется в соответствии с правами доступа для группы файла.
4. В остальных случаях доступ к файлу предоставляется в соответствии с правами доступа для остальных пользователей.

Проверки прав доступа для владельца, группы и остальных выполняются по порядку и прекращаются, как только обнаруживается подходящее правило. Это может привести к неожиданным последствиям: если, например, права доступа для группы превышают права владельца, то владелец будет фактически иметь меньше прав доступа к файлу, чем участники группы.

Если у файла установлен ACL, то применяется измененная версия алгоритма.

Для создания привилегированного процесса используются две привилегии: CAP_DAC_READ_SEARCH и CAP_DAC_OVERRIDE.

2.5. Изменение битов прав доступа к файлу

Для изменения битов прав доступа к файлу используется системный вызов **chmod()**:

```
#include <sys/stat.h>

int chmod(const char *pathname, mode_t mode);
```

Аргумент **mode** задает новые права доступа к файлу, указанному в аргументе **pathname**.

Аргумент **mode** можно записать как восьмеричное число или в виде суперпозиции, сформированной с помощью операции «ИЛИ» (|) из битов прав доступа, представленных в табл. 2.6.

В случае успеха возвращается 0. В случае ошибки возвращается -1, и в переменную **errno** записывается код ошибки.

Для изменения прав доступа к файлу необходимо, чтобы процесс обладал привилегией CAP_FOWNER, или чтобы его FSUID совпадал с идентификатором владельца файла UID.

Для повышения безопасности введены следующие ограничения:

- если процесс не обладает привилегией CAP_FSETID, и группа файла не совпадает с FSGID процесса или с одной из его дополнительных групп, бит **set-group-ID** будет сброшен, при этом данное действие не считается ошибкой;
- если процесс не обладает привилегией CAP_FSETID, и он пытается писать в файл, у которого установлены биты **set-user-ID** и/или **set-group-ID**, то эти биты будут сброшены.

Для изменения битов прав доступа к файлу предназначена команда **chmod**:

```
chmod [опции] права[,права] список_файлов
chmod [опции] права_восьмеричные список_файлов
chmod [опции] --reference=файл-образец список_файлов
```

Команда **chmod** может принимать права, которые нужно установить на файл, в трех видах: символьное выражение, восьмеричное число и файл-образец.

2.6. Пользовательская маска процесса

Для новых файлов ядро использует права, указанные в аргументе **mode** системного вызова **open()** или **creat()**. Для новых директорий эти права устанавливаются в соответствии с аргументом **mode** команды **mkdir()**. Однако указанные параметры изменяются с помощью пользовательской маски процесса **umask**, которая является атрибутом процесса. Она задает биты прав доступа, которые будут сброшены при создании процессом (обладающим данной маской) новых файлов или директорий.

Обычно процесс задействует атрибут **umask**, который он наследует от своей родительской оболочки. Поэтому пользователь может управлять данным атрибутом в программах, выполняемых из оболочки, используя встроенную в оболочку одноименную команду, изменяющую атрибут **umask** для процесса оболочки.

Файлы инициализации в большинстве оболочек по умолчанию устанавливают для атрибута **umask** восьмеричное значение 022 (----w--w-). Оно указывает на то, что право на запись должно быть всегда отключено для группы и для остальных пользователей.

Следовательно, если учесть, что обычно аргумент **mode** системного вызова **open()** равен 0666 (т.е. чтение и запись разрешены для всех пользователей), то новые файлы создаются с правами доступа на чтение и запись для владельца, а для группы и остальных – только с правом доступа на чтение (команда **ls –l** покажет это как rw-r--r--).

Аналогично для директорий, если учесть, что для аргумента **mode** системного вызова **mkdir()** установлено значение 0777 (т.е. все права доступа предоставлены всем пользователям), новые директории создаются с предоставлением всех прав доступа владельцу, а группам и остальным пользователям предоставляются только права доступа на чтение и выполнение (rwxr-xr-x) [9].

Для изменения пользовательской маски процесса предназначен системный вызов **umask()**:

```
#include <sys/stat.h>

mode_t umask(mode_t mask);
```

Аргумент **mask** задает новое значение пользовательской маски процесса. Его можно указывать как восьмеричное число или в виде строки, объединяющей с помощью операции «ИЛИ» (|) константы, приведенные в табл. 2.6.

Вызов **umask()** всегда завершается успешно и возвращает предыдущее значение маски.

Процесс-потомок, созданный системным вызовом **fork()**, наследует **umask** родителя. При выполнении системного вызова **execve()** значение **umask** не меняется.

Рассмотрим фрагмент программы **files/t_umask.c**, демонстрирующей использование системного вызова **umask()** [9]. В этой программе задается значение **umask**, а затем создаются обычный файл и директория. Программа выводит права, которые должны были быть назначены файлу и директории, и права, которые были назначены в итоге после применения **umask**.

Листинг 2.2. Фрагмент программы files/t_umask.c.

```
#define MYFILE "myfile"
#define MYDIR  "mydir"
#define FILE_PERMS     (S_IRUSR | S_IWUSR | S_IRGRP |
S_IWGRP)
#define DIR_PERMS      (S_IRWXU | S_IRWXG | S_IRWXO)
#define UMASK_SETTING (S_IWGRP | S_IXGRP | S_IWOTH |
S_IXOTH)

int main(int argc, char *argv[])
{
    int fd;
    struct stat sb;
    mode_t u;

/* Установить новую маску 033 */
    umask(UMASK_SETTING);

    fd = open(MYFILE, O_RDWR | O_CREAT | O_EXCL,
FILE_PERMS);
    if (fd == -1)
        errExit("open-%s", MYFILE);
    if (mkdir(MYDIR, DIR_PERMS) == -1)
        errExit("mkdir-%s", MYDIR);
```

```
/* Получить значение текущей маски и сбросить маску */
    u = umask(0);

    if (stat(MYFILE, &sb) == -1)
        errExit("stat-%s", MYFILE);
    printf("Requested file perms: %s\n", filePerm-
Str(FILE_PERMS, 0));
    printf("Process umask:       %s\n", filePermStr(u,
0));
    printf("Actual file perms:     %s\n\n", filePerm-
Str(sb.st_mode, 0));

    if (stat(MYDIR, &sb) == -1)
        errExit("stat-%s", MYDIR);
    printf("Requested dir. perms: %s\n", filePerm-
Str(DIR_PERMS, 0));
    printf("Process umask:       %s\n", filePermStr(u,
0));
    printf("Actual dir. perms:      %s\n", filePerm-
Str(sb.st_mode, 0));

    if (unlink(MYFILE) == -1)
        errMsg("unlink-%s", MYFILE);
    if (rmdir(MYDIR) == -1)
        errMsg("rmdir-%s", MYDIR);
    exit(EXIT_SUCCESS);
}
```

Запустим программу:

```
$ ./t_umask
Requested file perms: rw-rw----
Process umask:        ----wx-wx
Actual file perms:    rw-r-----

Requested dir. perms: rwxrwxrwx
Process umask:        ----wx-wx
Actual dir. perms:    rwxr--r--
```

Невозможно использовать вызов **umask()**, чтобы получить значение маски процесса без одновременного изменения маски. Поэтому необходимо второй раз вызывать umask(), чтобы восстановить

предыдущее (возвращенное) значение маски. Так как такой двойной вызов не является атомарным, может возникнуть состояние гонки в многопоточных программах.

Маску процесса можно также увидеть в поле **Umask** файла **/proc/<pid>/status**. Таким образом, процесс может узнать текущее значение своей маски, не изменяя ее значения. Например:

```
$ cat /proc/self/status | grep Umask
Umask:      0002
```

В BASH имеется встроенная команда **umask**:

```
umask [-p] [-S] [режим]
```

Если режим начинается с цифры, то он интерпретируется как восьмеричное число. В противном случае он интерпретируется как символьное выражение аналогично тому, как устанавливаются права доступа в команде **chmod**. Если режим опущен, то выводится текущее значение **umask**. Опция **-S** выводит значение прав, которые не затрагиваются маской в символьном виде. Опция **-p** выводит маску в виде команды, пригодной для использования в скриптах.

2.7. Флаги инода

Флаги инода позволяют связать с инодом дополнительные атрибуты, которые влияют на разные аспекты поведения файла, в том числе и на управление доступом к файлу.

Имена флагов и их числовые значения определяются в заголовочном файле ядра **fs/ext4/ext4.h**. Рассмотрим для примера несколько флагов, представленных в табл. 2.7.

В основном, если флаги файла установлены для директории, их автоматически наследуют новые файлы и директории, создаваемые в данной директории.

Для получения и изменения флагов файла используется системный вызов **ioctl()** с операциями FS_IOC_GETFLAGS и FS_IOC_SETFLAGS соответственно. Рассмотрим фрагмент программы [9], в которой выполняется установка флага FS_NOATIME_FL для файла, связанного с дескриптором **fd**.

Таблица 2.7

Некоторые флаги файла

Имя флага	Опция команды chattr	Описание
EXT4_ IMMUTABLE_FL	i	Файл защищен от изменений. Файл нельзя удалить или переименовать. Нельзя сделать новую ссылку на файл. Данные файла нельзя обновить (write() и truncate()), а изменения метаданных не допускаются (chmod(), chown(), unlink(), link(), rename(), rmdir(), utime(), setxattr() и removexattr()). Установить данный флаг могут только процессы с привилегией CAP_LINUX_IMMUTABLE. Когда флаг установлен, даже привилегированный процесс не может изменить содержимое файла или его метаданные
EXT4_APPEND_ FL	a	Файл можно открыть только для записи в режиме добавления в конец файла (должен быть установлен флаг O_APPEND). Этот флаг можно использовать, например, для файла журнала. Установить данный флаг могут только процессы с привилегией CAP_LINUX_IMMUTABLE
EXT4_NOATIME_ FL	A	У файла не обновляется время последнего доступа (поле **atime** инода). Это позволяет избежать обновления инода при каждом доступе к файлу и таким образом повышает эффективность ввода-вывода
EXT4_EXTENTS_ FL	e	Файл использует экстенты для хранения данных на диске

Листинг 2.3. Изменение флагов инода.

```
int attr;
/* Извлечь текущие флаги */
if (ioctl(fd, FS_IOC_GETFLAGS, &attr) == -1)
    errExit("ioctl");
/* Добавить новый флаг */
attr |= FS_NOATIME_FL;
/* Обновить флаги */
if (ioctl(fd, FS_IOC_SETFLAGS, &attr) == -1)
    errExit("ioctl");
```

Для изменения флагов инода необходимо, чтобы FSUID процесса соответствовал идентификатору владельца файла UID, либо чтобы процесс обладал привилегией CAP_FOWNER.

Для изменения флагов инода предназначена команда **chattr**:

```
chattr [опции] [режим] список_файлов
```

Формат режима следующий: +-=[aAcCdDeFijPsStTu]. Оператор '+' добавляет указанные флаги к существующим; оператор '-' удаляет указанные флаги; оператор '=' устанавливает указанные флаги в качестве единственных флагов инода.

Для просмотра флагов инода предназначена команда **lsattr**:

```
lsattr [опции] список_файлов
```

Рассмотрим пример с файлом, который нельзя изменить, в том числе удалить. Обратите внимание, что установить флаг **immutable** можно только в режиме суперпользователя (с помощью команды **sudo**).

```
$ touch f1
$ touch f2
$ ls -l
-rw-rw-r--. 1 user1 user1 0 Oct 10 05:02 f1
-rw-rw-r--. 1 user1 user1 0 Oct 10 05:02 f2
$ sudo chattr +i f1
$ lsattr
----i---------e----- ./f1
--------------e----- ./f2
```

```
$ rm f*
rm: cannot remove 'f1': Operation not permitted
$ ls -l
-rw-rw-r--. 1 user1 user1 0 Oct 10 05:02 f1
$ sudo rm f1
rm: cannot remove 'f1': Operation not permitted
```

Видно, что после установки флага **+i** файл не может удалить не только владелец файла, но и суперпользователь (с командой **sudo**). Для удаления такого файла нужно сначала снять флаг **immutable**.

Рассмотрим еще один пример с файлом, у которого установлен флаг **append only**, и в который можно только добавлять данные:

```
# echo abc > f_append_only
# echo 123456 > f_append_only
# ls -l f_append_only
-rw-r--r--. 1 root root 7 Oct 16 03:58 f_append_only
# cat f_append_only
123456
# chattr +a f_append_only
# lsattr f_append_only
-----a--------e----- f_append_only
# echo abc > f_append_only
bash: f_append_only: Operation not permitted
# echo abc >> f_append_only
# cat f_append_only
123456
abc
```

Видно, что после установки флага **+a** информацию в файл можно только добавлять. В BASH для этого нужно воспользоваться перенаправлением вывода ‘>>’.

2.8. Расширенные атрибуты файла

Расширенные атрибуты (extended attributes, EA) – это пары «имя-значение», которые можно связать с инодом файла. Они предназначены для расширения стандартных атрибутов дополнительными произвольными метаданными.

Расширенные атрибуты используются для хранения ACL, контекста безопасности SELinux и привилегий файла. Однако концепция расширенных атрибутов является настолько общей, что позволяет задействовать их и для других целей [9].

Имена расширенных атрибутов представлены в виде пары **namespace.name**, где **namespace** предназначен для разграничения расширенных атрибутов по функционально различным пространствам имен, а компонент **name** уникальным образом идентифицирует расширенный атрибут внутри данного пространства. Существует четыре пространства имен: **user**, **trusted**, **system** и **security**.

Расширенными атрибутами пространства **user** могут управлять непривилегированные процессы при определенных условиях: для извлечения значения данного атрибута необходимо иметь разрешение на чтение файла; для изменения значения данного атрибута необходимо иметь разрешение на запись. Расширенные атрибуты пространства **user** можно назначать только файлам и директориям. При этом, непривилегированному процессу не разрешено назначать расширенный атрибут **user** для директории, владельцем которой является другой пользователь, если для данной директории установлен **sticky**-бит.

Расширенные атрибуты реализованы на уровне файловой системы **ext4**. Для включения механизма расширенных атрибутов необходимо смонтировать файловую систему с ключом **user_xattr**:

```
$ mount -o user_xattr [устройство] [точка_монтирования]
```

Расширенными атрибутами пространства **trusted** могут управлять процессы, обладающие привилегией CAP_SYS_ADMIN.

Расширенные атрибуты пространства **system** используются ядром для хранения списка контроля доступа (ACL) файла.

Расширенные атрибуты пространства **security** используются ядром для хранения контекстов безопасности SELinux, а также привилегий исполняемых файлов.

Для установки расширенных атрибутов файла используется команда **setfattr**:

```
setfattr [опции] -n имя [-v значение] список_файлов
setfattr [опции] -x имя список_файлов
```

Опция **-n** устанавливает значение атрибута, опция **-x** полностью удаляет атрибут. Атрибут может быть не определен; значение расширенного атрибута может быть пустой строкой.

Для просмотра расширенных атрибутов файла используется команда **getfattr**:

```
getfattr [опции] -n имя список_файлов
getfattr [опции] -d [-m шаблон] список_файлов
```

Опция **-n** выводит значение атрибута по его имени, опция **-d** (--dump) выводит значение атрибутов, имена которых удовлетворяют шаблону. Опция **-m** задает шаблон в виде регулярного выражения. Шаблон по умолчанию соответствует всем атрибутам в пространстве **user**. Шаблон '-' включает все атрибуты.

Для установки расширенного атрибута предназначен системный вызов **setxattr()**.

```
#include <sys/xattr.h>

int setxattr(const char *path, const char *name,
    const void *value, size_t size, int flags);
```

Аргумент **path** задает путь к файлу. Аргумент **name** – имя атрибута, которое должно завершаться нулем. Аргумент **value** – указатель на буфер, содержащий новое значение атрибута. Аргумент **size** задает размер значения атрибута в байтах, т.е. длину буфера; можно задать нулевую длину. По умолчанию, если аргумент **flags** равен нулю, атрибут будет создан, если он не существует, или значение будет обновлено, если атрибут существует.

Для удаления расширенного атрибута предназначен системный вызов **removexattr()**.

```
#include <sys/xattr.h>

int removexattr(const char *path, const char *name);
```

Аргумент **path** задает путь к файлу. Аргумент **name** – имя удаляемого атрибута.

В случае успеха возвращается 0. В случае ошибки возвращается -1, и в переменную **errno** записывается код ошибки.

Для получения значения расширенного атрибута предназначен системный вызов **getxattr()**.

```
#include <sys/xattr.h>

ssize_t getxattr(const char *path, const char *name,
                 void *value, size_t size);
```

Аргумент **path** задает путь к файлу. Аргумент **name** – имя извлекаемого атрибута. Значение атрибута возвращается в буфер, на который указывает аргумент **value**. Длина буфера указывается в аргументе **size**. В случае успеха возвращается количество байтов, скопированное в аргумент **value**.

Если указать для аргумента **size** значение 0, то системный вызов вернет размер значения атрибута.

Для получения списка всех расширенных атрибутов предназначен системный вызов **listxattr()**.

```
#include <sys/xattr.h>

ssize_t listxattr(const char *path, char *list, size_t
size);
```

Аргумент **path** задает путь к файлу. Список имен расширенных атрибутов возвращается в виде последовательности строк с завершающим нулем в буфер, на который указывает аргумент **list**. Размер данного буфера должен быть задан в аргументе **size**. В случае успеха возвращается количество байтов, скопированных в список **list**. Если указать для аргумента **size** значение 0, то системный вызов вернет размер списка.

В целях повышения безопасности из перечня расширенных атрибутов в списке **list** могут быть исключены атрибуты, для доступа к которым у вызывающего процесса нет прав. Например, многие системы не включают атрибуты **trusted** в список, возвращаемый системным вызовом **listxattr()**, выполненным непривилегированным процессом. Поэтому, необходимо допускать возможность того, что при последующем системном вызове **getxattr()**, использующем имя

расширенного атрибута из списка **list**, может произойти ошибка, поскольку данный процесс не обладает привилегиями, необходимыми для получения значения этого атрибута. Подобная ошибка могла бы произойти также в том случае, если бы другой процесс удалил какой-либо атрибут между моментами вызовов **listxattr()** и **getxattr()** [9].

2.9. Пример работы с расширенными атрибутами файла

Рассмотрим пример программы, приведенной в справочной странице **listxattr(2)**. В программе показано использование системных вызовов **listxattr()** и **getxattr()**. Для файла, путь к которому передается через первый аргумент командной строки, программа выводит расширенные атрибуты и их значения.

Листинг 2.4. Программа p_listxattr.c.

```
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/xattr.h>

int
main(int argc, char *argv[])
{
    ssize_t buflen, keylen, vallen;
    char *buf, *key, *val;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s path\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    /*
     * Определяем необходимую длину буфера, в который
запишем имена расширенных атрибутов.
     */
    buflen = listxattr(argv[1], NULL, 0);
    if (buflen == -1) {
        perror("listxattr");
```

```
    exit(EXIT_FAILURE);
}
if (buflen == 0) {
    printf("%s has no attributes.\n", argv[1]);
    exit(EXIT_SUCCESS);
}

/*
 * Выделяем память под буфер.
 */
buf = malloc(buflen);
if (buf == NULL) {
    perror("malloc");
    exit(EXIT_FAILURE);
}

/*
 * Копируем список имен атрибутов в буфер.
 */
buflen = listxattr(argv[1], buf, buflen);
if (buflen == -1) {
    perror("listxattr");
    exit(EXIT_FAILURE);
}

/*
 * Проходим в цикле по списку имен атрибутов
 * (строки, оканчивающиеся нулем). Используем
 * оставшуюся длину буфера
 * для определения конца списка.
 */
key = buf;
while (buflen > 0) {

    /*
     * Выводим имя атрибута.
     */
    printf("%s: ", key);

    /*
     * Определяем длину значения атрибута.
     */
```

```
        vallen = getxattr(argv[1], key, NULL, 0);
        if (vallen == -1)
            perror("getxattr");

        if (vallen > 0) {

            /*
             * Выделяем память под значения атрибута.
             * Нужен один дополнительный байт, чтобы
дополнить 0x00.
             */
            val = malloc(vallen + 1);
            if (val == NULL) {
                perror("malloc");
                exit(EXIT_FAILURE);
            }

            /*
             * Копируем значение атрибута в буфер.
             */
            vallen = getxattr(argv[1], key, val, val-
len);
            if (vallen == -1)
                perror("getxattr");
            else {
                /*
                 * Выводим значение атрибута.
                 */
                val[vallen] = 0;
                printf("%s", val);
            }

            free(val);
        } else if (vallen == 0)
            printf("<no value>");

        printf("\n");

        /*
         * Берем имя следующего атрибута.
         */
        keylen = strlen(key) + 1;
```

```
        buflen -= keylen;
        key += keylen;
    }

    free(buf);
    exit(EXIT_SUCCESS);
}
```

Создадим тестовый файл и установим для него два расширенных атрибута в пространстве **user**: один непустой и один пустой. Кроме того, как видно, с файлом связан контекст безопасности SELinux, установленный автоматически при создании файла. Как уже было сказано, ACL, контекст безопасности SELinux и привилегии файла хранятся в его расширенных атрибутах.

```
$ touch f1
$ setfattr -n user.university -v MEPhI f1
$ setfattr -n user.empty f1
$ ls -Z f1
unconfined_u:object_r:user_home_t:s0 f1
```

Скомпилируем и запустим программу, передав ей в качестве аргумента имя тестового файла:

```
$ gcc p_listxattr.c -o p_listxattr
$ ./p_listxattr f1
security.selinux:unconfined_u:object_r:user_home_t:s0
user.empty: <no value>
user.university: MEPhI
```

Программа вывела как расширенные атрибуты, установленные нами явно, так и контекст безопасности SELinux, установленный автоматически при создании файла. Обратите внимания на пространства, в которых хранятся те или иные расширенные атрибуты.

2.10. Списки контроля доступа файла

Список контроля доступа (Access Control Lists, ACL) представляет собой набор записей, каждая из которых задает права доступа к

файлу для конкретного пользователя или конкретной группы пользователей. ACL расширяет дискреционную модель доступа к файлам, позволяя задавать индивидуальные права достаточно большому количеству пользователей и групп.

Механизм ACL реализован на уровне файловой системы **ext4**. Для включения механизма ACL необходимо смонтировать файловую систему с ключом **acl**:

```
$ mount -o acl [устройство] [точка_монтирования]
```

Существует два типа ACL:

- ACL, который используется для определения прав доступа процесса к файлу. Такие ACL связываются с обычными файлами, директориями и другими типами файлов. Для их хранения используется расширенный атрибут **system.posix_acl_access**. Будем в тексте такие ACL называть просто ACL;
- ACL по умолчанию, который связывается только с директориями и определяет начальный ACL, который связывается с файлами и директориями, создаваемыми в данной директории. Для их хранения используется расширенный атрибут **system.posix_acl_default**. Будем в тексте такие ACL называть ACL по умолчанию.

Процесс может прочитать ACL файла, только если у него есть право поиска в директории, содержащей файл.

2.11. Формат записей ACL

Каждая запись состоит из трех частей:

- тип записи;
- идентификатор пользователя или группы (UID или GID) в записях типа ACL_USER или ACL_GROUP;
- права доступа как комбинация чтения, записи и исполнения/поиска.

Возможные типы записей представлены в табл. 2.8.

Таблица 2.8

Типы записей ACL

Тип записи	Назначение
ACL_USER_OBJ	Запись определяет права доступа для владельца файла. ACL должен содержать ровно одну такую запись. Она задает традиционные права доступа для владельца (пользователя) файла
ACL_USER	Запись определяет права доступа для пользователя, идентификатор которого содержит запись. Таких записей может быть от нуля и больше, однако для конкретного пользователя не может быть более одной такой записи
ACL_GROUP_OBJ	Запись определяет права доступа для группы – владельца файла. ACL должен содержать ровно одну такую запись. Она задает традиционные права доступа для группы – владельца файла, если только ACL не содержит запись ACL_MASK
ACL_GROUP	Запись определяет права доступа для группы, идентификатор которой содержит запись. Таких записей может быть от нуля и больше, однако для конкретной группы не может быть более одной такой записи
ACL_MASK	Запись определяет максимальные права доступа, которые могут быть предоставлены записями ACL_USER, ACL_GROUP_OBJ, ACL_GROUP. ACL содержит не более одной записи ACL_MASK. Если ACL содержит записи ACL_USER или ACL_GROUP, то наличие записи ACL_MASK является обязательным. Она задает традиционные права доступа для группы – владельца файла
ACL_OTHER	Запись определяет права доступа для процессов, которые не соответствуют другим записям в ACL. ACL должен содержать ровно одну такую запись. Она задает традиционные права доступа для остальных пользователей

На рис. 2.2 показано, как соотносятся категории пользователей, используемые битами прав доступа, и типы записей ACL.

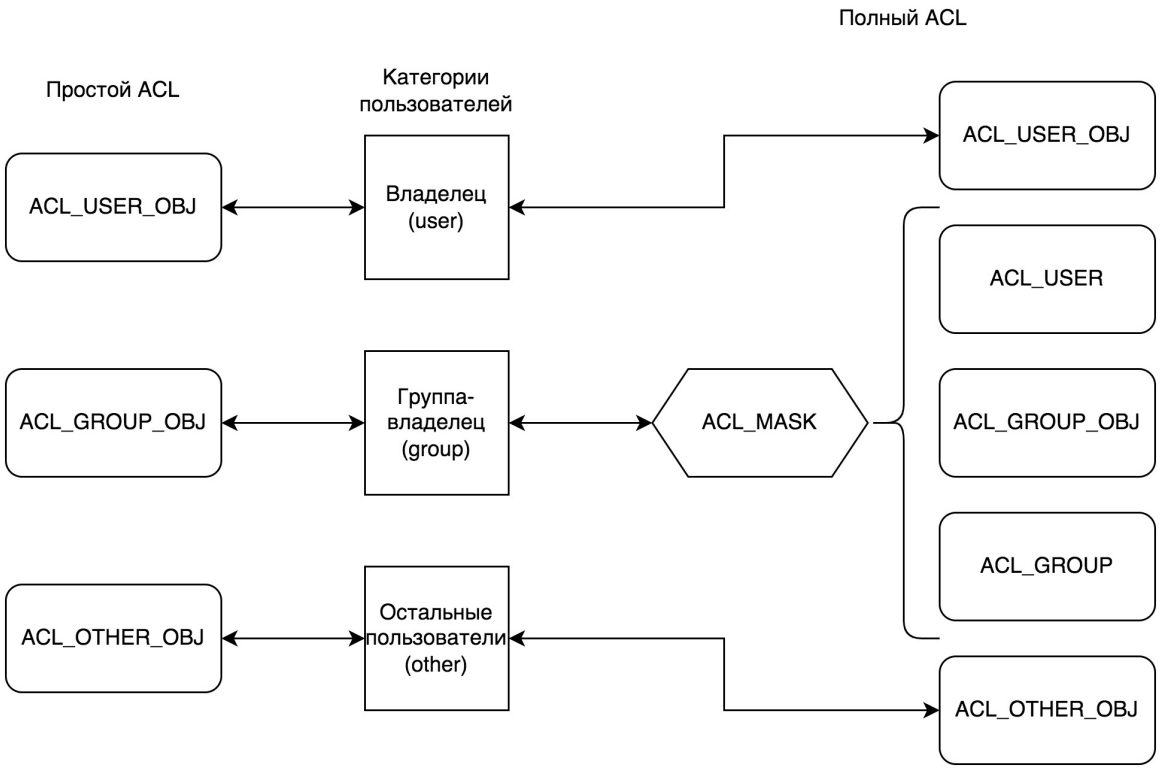


Рис. 2.2. Типы записей ACL

2.12. Алгоритм проверки прав доступа с помощью ACL

Проверка производится в следующем порядке до первого совпадения:

1. Если процесс является привилегированным, то предоставляются все права, за исключением случая, когда привилегированный процесс запускает исполняемый файл. Такому процессу предоставляется право на исполнение, только если оно предоставлено хотя бы одной записью ACL.

2. Если FSUID процесса совпадает с идентификатором владельца файла UID, то процессу предоставляются права доступа, указанные в записи ACL_USER_OBJ.

3. Если FSUID процесса совпадает с идентификатором какой-либо записи ACL_USER, то данному процессу предоставляются права, указанные в этой записи, логически умноженные (операция «И») со значением ACL_MASK.

4. Если FSGID или любой идентификатор дополнительной группы процесса совпадает с идентификатором группы – владельца файла GID (запись ACL_GROUP_OBJ) или с идентификатором какой-либо записи ACL_GROUP, и найденная запись предоставляет требуемые права, данному процессу предоставляются права, указанные в этой записи, логически умноженные (операция «И») со значением ACL_MASK. Если такая запись найдена, но она не предоставляет требуемые права, то доступ запрещается.

5. В остальных случаях процессу предоставляются права доступа, указанные в записи ACL_OTHER.

2.13. Назначение записи ACL_MASK

Запись ACL_MASK ограничивает права доступа, предоставляемые записями ACL_USER, ACL_GROUP и ACL_GROUP_OBJ. Если ACL содержит хотя бы одну запись ACL_USER или ACL_GROUP, то он должен содержать запись ACL_MASK. Если записей ACL_USER или ACL_GROUP нет, то запись ACL_MASK является необязательной.

С позиции рассмотрения механизма ACL программы можно разделить на две группы:

- программы, использующие механизм ACL (ACL-aware), которые скомпилированы с библиотекой **libacl** (можно посмотреть с помощью команды **ldd**);
- программы, не использующие механизм ACL.

Запись ACL_MASK предназначена для обеспечения совместимости механизма ACL с программами, которые его не используют. Она позволяет выполнять вызов **chmod()** в отношении файлов, с которыми связаны ACL, не затрагивая записей ACL_GROUP и ACL_GROUP_OBJ.

Если ACL содержит запись ACL_MASK:

- все изменения в традиционных правах доступа группы с помощью системного вызова **chmod()** изменяют права записи ACL_MASK (а не записи ACL_GROUP_OBJ);
- вызов **stat()** возвращает права доступа ACL_MASK (а не ACL_GROUP_OBJ) в битах прав доступа для группы в поле **st_mode**.

2.14. ACL по умолчанию

Если у директории есть ACL по умолчанию, то:

- директория, создаваемая в данной директории, наследует ACL по умолчанию в качестве своего ACL по умолчанию;
- новый файл или директория, создаваемые в данной директории, устанавливают ACL по умолчанию в качестве своего ACL доступа. Для записей соответствующих традиционным битам прав доступа (ACL_USER_OBJ, ACL_MASK или, если отсутствует запись ACL_MASK, то ACL_GROUP_OBJ, ACL_OTHER) применяется операция «И» с соответствующими битами аргумента mode системного вызова (**open()**, **mkdir()**), использованного для создания файла или поддиректории.

Если у директории есть ACL по умолчанию, то маска **umask** не принимает участия в определении прав доступа для записей ACL доступа нового файла, созданного в данной директории.

Если у директории нет ACL по умолчанию, то:

- новые поддиректории также не имеют ACL по умолчанию;
- права доступа нового файла или директории устанавливаются по традиционным правилам: права устанавливаются в значение

аргумента **mode** (переданного в **open()**, **mkdir()**), исключая биты из маски процесса.

2.15. Командный интерфейс

Если на файл установлен ACL, то команда **ls -l** выведет минимальную информацию об ACL. Знак ’+’ в конце строки битов доступа говорит о том, что с файлом связан ACL. Права для владельца файла совпадают с правами владельца файла, права для группы-владельца показывают установленную ACL-маску, а не права для группы – владельца файла. Права для остальных пользователей совпадают с правами для остальных пользователей.

Если на файл установлен ACL, то с помощью команды **chmod** нельзя изменить права для группы – владельца файла, вместо этого будет изменена ACL-маска. Чтобы изменить права для группы – владельца файла, нужно использовать команду **setfacl -m g::права имя_файла**.

Для вывода ACL предназначена команда **getfacl**.

```
getfacl [опции] список_файлов
getfacl [опции] -
```

По умолчанию **getfacl** выводит как ACL доступа, так и ACL по умолчанию (если это директория, у которой есть ACL по умолчанию), а также в виде комментариев эффективные права для тех записей, права в которых отличаются от эффективных. Опции меняют это поведение. Опция **-a** выводит только ACL доступа, а опция **-d** выводит только ACL по умолчанию. Опция **-R** выводит ACL рекурсивно для всех файлов и директорий. Если вместо списка файлов написать знак ‘-’, то **getfacl** читает список файлов из стандартного ввода.

Вывод команды **getfacl** может быть использован как ввод для команды **setfacl**. Например, можно с помощью команды **getfacl -R /путь_к_директории** вывести все ACL для директории и ее содержимого. Этот вывод можно сохранить в файле и затем использовать для массового восстановления ACL на иерархии файлов с помощью команды **setfacl --set-file=имя_файла**.

Для каждого файла выводится следующая информация.

Первые три строки в виде комментариев содержат имена файла, владельца и группы-владельца.

Четвертая строка содержит дополнительные права доступа: set-user-ID (s), set-group-ID (s), sticky (t) или (-), если право отсутствует. Если все права отсутствуют, строка не выводится.

Следующие строки выводят ACL доступа и по порядку содержат права для владельца, права для именованных пользователей, права для группы-владельца, права для именованных групп. Далее выводится маска (mask), которая ограничивает права, назначенные именованным пользователям и всем группам (права владельца и всех остальных пользователей не затрагиваются маской). Затем выводятся права для всех остальных пользователей системы (other).

В отличие от обычных файлов, директории могут иметь ACL по умолчанию, который определяет права на вновь создаваемые файлы и директории.

Следующие строки выводят ACL по умолчанию в том же порядке, что и ACL доступа. Обычно для пользователей и групп присутствует право x, которое для обычных файлов будет проигнорировано (так как новые файлы обычно создаются неисполняемыми), а для директорий будет установлено право на поиск.

Для установки ACL предназначена команда **setfacl**:

```
setfacl [опции] [{-m|-x} acl_spec] [{-M|-X} acl_file] файл
setfacl --restore=файл
```

2.16. Выполнение ввода/вывода

Для работы с файлом процесс использует дескриптор открытого файла. Процесс обращается к дескриптору по его номеру в таблице открытых файлов процесса. Чтобы выполнить ввод/вывод в отношении обычного файла, сначала нужно получить его дескриптор с помощью системного вызова **open()**. Затем ввод/вывод выполняется с помощью системных вызовов **read()** и **write()**. После завершения всех операций следует освободить дескриптор файла и связанные с ним ресурсы с помощью системного вызова **close()**.

Обычно процесс запускается с тремя стандартными дескрипторами открытых файлов, представленных в табл. 2.9. Эти три дескриптора открыты командной оболочкой BASH, и при запуске программа наследует у процесса BASH копии дескрипторов файлов. Если в командной строке указано перенаправление ввода/вывода, то BASH перед запуском программы обеспечивает соответствующее изменение дескрипторов файлов.

Таблица 2.9

Стандартные дескрипторы открытых файлов

Номер	Назначение	Имя в POSIX	Поток stdio
0	Стандартный ввод	STDIN_FILENO	stdin
1	Стандартный вывод	STDOUT_FILENO	stdout
2	Стандартная ошибка	STDERR_FILENO	stderr

При ссылке на эти дескрипторы файлов в программе можно использовать либо номера (0, 1 или 2), либо стандартные имена POSIX, определенные в файле **<unistd.h>**.

Исторически в UNIX используется концепция «все есть файл», т.е. модель ввода/вывода является универсальной. Это означает, что одни и те же четыре системных вызова – **open()**, **read()**, **write()** и **close()** – применяются для выполнения ввода/вывода для всех типов файлов, за исключением директорий.

Универсальность ввода/вывода достигается обеспечением того, что в каждой файловой системе и в каждом драйвере устройства реализуется один и тот же набор системных вызовов ввода/вывода. Поскольку детали реализации конкретной файловой системы или устройства скрывает ядро, при написании программ разработчик может игнорировать специфику устройства.

2.17. Открытие файла

Системный вызов **open()** либо открывает существующий файл, либо создает и открывает новый файл.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pathname, int flags, mode_t mode);
```

Аргумент **pathname** задает путь к файлу. Если в этом аргументе находится символьная ссылка, она преобразуется в новый путь к файлу. В случае успеха **open()** возвращает номер дескриптора открытого файла, который используется для работы с файлом в последующих системных вызовах. В случае ошибки **open()** возвращает –1, а для **errno** устанавливается код ошибки.

Аргумент **flags** является битовой строкой, указывающей режим доступа к файлу с использованием одной из констант, перечисленных в табл. 2.10.

Таблица 2.10

Значения флагов системного вызова **open()**

Флаг	Описание
Флаги режима доступа к файлу	
O_RDONLY	Открытие файла только для чтения
O_WRONLY	Открытие файла только для записи
O_RDWR	Открытие файла как для чтения, так и для записи
Флаги создания файла	
O_CLOEXEC	Установка флага закрытия при выполнении (close-on-exec)
O_CREAT	Создание файла, если он еще не существует
O_DIRECTORY	Аргумент **pathname** должен указывать на директорию, иначе ошибка
O_EXCL	С флагом O_CREAT: исключительное создание файла

Окончание табл. 2.10

Флаг	Описание
O_NOCTTY	**pathname** запрещено становиться управляющим терминалом данного процесса
O_NOFOLLOW	Запрет на разыменование символьных ссылок
O_TMPFILE	Создание временного файла без имени. Аргумент **pathname** должен указывать на директорию, в которой будет создан файл. Все, что будет записано в файл, будет утеряно после закрытия последнего дескриптора файла
O_TRUNC	Усечение существующего файла до нулевой длины
Флаги ввода/вывода	
O_APPEND	Записи добавляются исключительно в конец файла
O_ASYNC	Генерация сигнала, когда возможен ввод/вывод
O_DIRECT	Операции ввода/вывода осуществляются без использования кэша
O_DSYNC	Синхронизированный ввод/вывод с обеспечением целостности данных
O_NOATIME	Запрет на обновление времени последнего доступа к файлу при чтении с помощью системного вызова **read()**
O_NONBLOCK	Открытие в неблокируемом режиме
O_SYNC	Ведение записи в файл в синхронном режиме

Если создается новый файл, то аргумент **mode** должен содержать права доступа, которые будут присвоены файлу (см. табл. 2.6). Если не указаны ни флаг O_CREAT, ни флаг O_TMPFILE, то аргумент **mode** игнорируется.

Права доступа, присваиваемые новому файлу, зависят не только от аргумента **mode**, но и от значения маски (umask) процесса и от ACL по умолчанию родительской директории, если он установлен.

В Linux информацию о дескрипторах открытых файлов процесса можно получить в директории **/proc/<pid>/fd**, а в директории **/proc/<pid>/fdinfo** находится по одному файлу для каждого открытого процессом файла с именем, совпадающим с номером дескриптора. В поле **pos** этого файла показано текущее смещение в файле. В поле **flags** находится восьмеричное число, показывающее флаги файла.

Чтобы вывести в командной строке файлы, открытые процессом в настоящее время, можно воспользоваться помощью команд **lsof** или **fuser**.

В случае ошибки возвращается -1, и код ошибки записывается в переменную **errno**.

Исторически в UNIX для создания нового файла использовался системный вызов **creat()**, который эквивалентен вызову **open()** со следующими флагами:

```
fd = open(pathname, O_CREAT | O_WRONLY | O_TRUNC, mode);
```

В Linux системный вызов **creat()** считается устаревшим.

2.18. Чтение из файла

Системный вызов **read()** позволяет считывать данные из открытого файла, на который ссылается дескриптор **fd**.

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
```

Аргумент **count** определяет количество считываемых байтов, которые нужно сохранить в буфере памяти по адресу из аргумента **buf**. Буфер **buf** должен иметь длину в байтах не менее той, что задана в аргументе **count**. Обратите внимание, что память под буфер нужно выделить заранее и в системный вызов нужно передать именно указатель на буфер.

В случае успеха возвращается количество прочитанных байт или 0, если встретился символ конца файла. В случае ошибки возвращается -1. Поэтому для возвращаемого значения используется тип данных **ssize_t**, который является целочисленным типом со знаком.

Чтение из терминала выполняется до первого встреченного символа новой строки (‘\n’).

Директории обладают внутренней структурой, поэтому для их чтения предназначен отдельный системный вызов **getdents()**.

```
int getdents(unsigned int fd, struct linux_dirent
*dirp,
                    unsigned int count);
int getdents64(unsigned int fd, struct linux_dirent64
*dirp,
                    unsigned int count);
```

Системный вызов **getdents()** читает несколько структур **linux_dirent** из директории, на которую ссылается дескриптор открытого файла **fd**, в буфер **dirp** размером **count**. Структура **linux_dirent** объявлена следующим образом:

```
struct linux_dirent {
    unsigned long  d_ino;     /* Номер инода */
    unsigned long  d_off;     /* Смещение от начала
директории до начала следующей linux_dirent */
    unsigned short d_reclen;  /* Размер этой
linux_dirent */
    char           d_name[];  /* Имя файла, заканчи-
вающееся null */
/* Длина вычисляется как (d_reclen - 2 -
offsetof(struct linux_dirent, d_name)) */
/*
    char           pad;       // Дополняющий байт
    char           d_type;    // Тип файла. Смещение
(d_reclen - 1)
*/
}
```

Файловые системы семейства **ext** хранят в записях директории тип файла для повышения производительности.

Системный вызов **getdents64()** поддерживает большие файловые системы. Его синтаксис аналогичен вызову **getdents()**, за исключением того, что его второй аргумент является указателем на буфер, содержащий структуры следующего вида:

```
struct linux_dirent64 {
    ino64_t        d_ino;    /* 64-битный номер инода
*/
    off64_t        d_off;    /* 64-битное смещение до
следующей структуры */
    unsigned short d_reclen; /* Размер этой dirent */
    unsigned char  d_type;   /* Тип файла */
    char           d_name[]; /* Имя файла, заканчива-
ющееся null */
};
```

В случае успеха возвращается количество прочитанных байт. При достижении конца директории возвращается 0. В случае ошибки возвращается -1, и код ошибки записывается в переменную **errno**.

Рассмотрим пример программы, приведенной в справочной странице **getdents(2)**, выводящей на экран содержимое директории.

Листинг 2.5. Программа p_dir.c.

```
#define _GNU_SOURCE
#include <dirent.h>
#include <fcntl.h>
#include <stdint.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/syscall.h>

#define handle_error(msg) \
    do { perror(msg); exit(EXIT_FAILURE); } while (0)

struct linux_dirent {
    unsigned long  d_ino;
    off_t          d_off;
    unsigned short d_reclen;
```

```
    char           d_name[];
};

#define BUF_SIZE 1024

int
main(int argc, char *argv[])
{
    int fd;
    long nread;
    char buf[BUF_SIZE];
    struct linux_dirent *d;
    char d_type;

    fd = open(argc > 1 ? argv[1] : ".", O_RDONLY |
O_DIRECTORY);
    if (fd == -1)
        handle_error("open");

    for (;;) {
        nread   =   syscall(SYS_getdents,   fd,   buf,
BUF_SIZE);
        if (nread == -1)
            handle_error("getdents");

        if (nread == 0)
           break;

        printf("--------------- nread=%d ------------
---\n", nread);
        printf("inode#    file type  d_reclen  d_off
d_name\n");
        for (long bpos = 0; bpos < nread;) {
            d = (struct linux_dirent *) (buf + bpos);
         printf("%8ld  ", d->d_ino);
         d_type = *(buf + bpos + d->d_reclen - 1);
         printf("%-10s ", (d_type == DT_REG) ?  "reg-
ular" :
                        (d_type  ==  DT_DIR)  ?   "direc-
tory" :
                        (d_type == DT_FIFO) ? "FIFO" :
                        (d_type == DT_SOCK) ? "socket" :
```

```
                        (d_type == DT_LNK) ?  "symlink" :
                        (d_type  ==  DT_BLK)  ?    "block
dev" :
                        (d_type  ==  DT_CHR)  ?    "char
dev" : "???");
          printf("%4d %10jd  %s\n", d->d_reclen,
                (intmax_t) d->d_off, d->d_name);
          bpos += d->d_reclen;
        }
    }

    exit(EXIT_SUCCESS);
}
```

Скомпилируем и запустим программу на примере небольшой директории. Затем сравним вывод программы с аналогичным выводом команды **ls** в длинном неотсортированном формате:

```
$ gcc p_dir.c -o p_dir
$ ./p_dir /etc/selinux/
--------------- nread=192 ---------------
inode#    file type  d_reclen d_name
 5505163  directory        24 .
 5506365  directory        32 targeted
 5507100  regular          32 config
 5506406  regular          40 .config_backup
 5505025  directory        24 ..
 5507101  regular          40 semanage.conf
$ ls -laU /etc/selinux/
total 32
drwxr-xr-x.   3 root root  4096 Jan 13  2021 .
drwxr-xr-x.   5 root root  4096 Jan  3 18:11 targeted
-rw-r--r--.   1 root root   548 Apr 23  2020 config
-rw-r--r--.    1 root root    576 Apr 22   2021 .con-
fig_backup
drwxr-xr-x. 146 root root 12288 Jan 20 21:56 ..
-rw-r--r--.    1 root root   2446 Jan 29   2020 sem-
anage.conf
```

Видно, что порядок записей в директории совпадает.

2.19. Запись в файл

Системный вызов **write()** записывает данные в открытый файл:

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);
```

Аргумент **buf** представляет собой адрес записываемых данных, **count** является количеством записываемых из буфера данных, а **fd** содержит дескриптор файла, который ссылается на тот файл, куда будут записываться данные.

В случае успеха вызов **write()** возвращает количество фактически записанных данных, которое может быть меньше значения аргумента **count**. Для дискового файла возможной причиной такой частичной записи может оказаться переполнение диска или достижение ограничения ресурса процесса на размеры файла (RLIMIT_FSIZE).

2.20. Закрытие файла

Системный вызов **close()** закрывает дескриптор файла, освобождая его для последующего повторного использования процессом. Когда процесс прекращает работу, все его дескрипторы открытых файлов автоматически закрываются:

```
#include <unistd.h>

int close(int fd);
```

Считается хорошей практикой явно закрывать неиспользуемые дескрипторы файлов.

2.21. Пример работы с файловым вводом/выводом

Рассмотрим фрагмент программы **fileio/copy.c** [9]. Эта программа является упрощенной версией команды **cp(1)**; она копирует содер-

жимое существующего файла, имя которого указано в первом аргументе командной строки, в новый файл с именем, указанным во втором аргументе командной строки.

Листинг 2.6. Фрагмент программы fileio/copy.c.

```
#define BUF_SIZE 1024

int main(int argc, char *argv[])
{
    int inputFd, outputFd, openFlags;
    mode_t filePerms;
    ssize_t numRead;
    char buf[BUF_SIZE];

    if (argc != 3 || strcmp(argv[1], "--help") == 0)
        usageErr("%s old-file new-file\n", argv[0]);

    /* Открытие файлов для ввода и для вывода */

    inputFd = open(argv[1], O_RDONLY);
    if (inputFd == -1)
        errExit("opening file %s", argv[1]);

    openFlags = O_CREAT | O_WRONLY | O_TRUNC;
    filePerms = S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP
|
                S_IROTH | S_IWOTH;      /* rw-rw-rw-
*/
    outputFd = open(argv[2], openFlags, filePerms);
    if (outputFd == -1)
        errExit("opening file %s", argv[2]);

    /* Перемещение данных до достижения конца файла
ввода или возникновения ошибки */

    while ((numRead = read(inputFd, buf, BUF_SIZE)) >
0)
        if (write(outputFd, buf, numRead) != numRead)
            fatal("write() returned error or partial
write occurred");
    if (numRead == -1)
        errExit("read");
```

```
    if (close(inputFd) == -1)
        errExit("close input");
    if (close(outputFd) == -1)
        errExit("close output");

    exit(EXIT_SUCCESS);
}
```

Контрольные вопросы

1. Где хранятся атрибуты (метаданные) файла?
2. Где хранится имя файла?
3. Где хранится содержимое файла? Какие особенности хранения содержимого обычного файла и директории?
4. Какие условия влияют на назначение группы – владельца нового файла?
5. Какими дискреционными правами должен обладать процесс, чтобы удалить файл?
6. На один файл (инод) имеется две жесткие ссылки. Как определить, какая ссылка была создана раньше, а какая позже?
7. В каком пространстве расширенных атрибутов хранится контекст безопасности файла?
8. Как опции монтирования файловой системы влияют на работу с файлами?
9. Какие типы файлов могут иметь ACL доступа, а какие могут иметь ACL по умолчанию?
10. Для чего предназначена запись ACL_MASK?

Глава 3. Управление доступом к процессам

В данной главе рассматривается понятие процесса и его жизненный цикл. Описывается механизм привилегий, представляющий важное значение с точки зрения безопасности операционной системы.

3.1. Определение процесса

Концепция процесса является фундаментальной для многозадачной операционной системы. Самое простое определение процесса следующее: *процесс* – это программа во время исполнения. На диске хранится исполняемый файл, который представляет собой скомпилированную программу, и содержимым которого является двоичный код. Когда эту программу запускают на исполнение, она становится процессом. Говорят, что программа выполняется в контексте процесса.

Чтобы запустить программу, сначала нужно считать ее содержимое в оперативную память, т.е. выделить место для этой программы, затем нужно выделить время CPU для выполнения инструкций программы. В ходе выполнения программа может открывать файлы и записывать туда информацию, взаимодействовать с другими работающими программами, устанавливать сетевые соединения и т.д. Другими словами, для своего выполнения программа требует выделения системных ресурсов (места в RAM, времени CPU, пропускной способности системы ввода/вывода и сетевой системы и т.д.). Как уже было сказано, управлением ресурсами занимается ядро операционной системы. Поэтому, рассматривая ядро, будем говорить, что *процесс* – это *единица потребления системных ресурсов*, такая сущность, для которой выделяются системные ресурсы. Т.е. задача обеспечения программы необходимыми ресурсами решается как задача выделения ресурсов процессу, а также их учета и мониторинга.

Процесс можно рассматривать как некий контейнер, в который загружается программа. Несколько процессов могут выполнять одну и ту же программу *одновременно*, при этом один процесс может выполнять несколько программ *последовательно*.

Ядро Linux является многопроцессным (или многозадачным) в том смысле, что одновременно на одном процессоре могут выполняться несколько процессов, чередуя свое выполнение во времени. За справедливым распределением времени CPU между конкурирующими процессами следит *планировщик* (scheduler), который выбирает для выполнения наиболее приоритетный процесс, а также может прервать (вытеснить) выполнение процесса, если появился более приоритетный процесс. Такой подход к планированию называется *вытесняющей* (preemptable) многозадачностью.

В понятие процесса включается:

- адресное пространство (address space), выделенное процессу в памяти, в которое загружаются инструкции программы;
- контекст (состояние) процесса;
- атрибуты безопасности;
- окружение (environment) процесса;
- выделенные системные ресурсы (например, дескрипторы открытых файлов и сетевые порты).

3.2. Дескриптор процесса

Чтобы управлять процессами, ядру нужна информация об их состоянии, связях друг с другом и взаимодействии с разными подсистемами ядра.

Для описания процесса ядро использует *дескриптор процесса*, который представлен структурой **task_struct**. В табл. 3.1 показаны основные поля дескриптора процесса.

Таблица 3.1

Основные поля структуры **task_struct**

Атрибут	**Назначение**
state	Состояние выполняющегося процесса
flags	Флаги процесса
*sched_class	Класс планирования

Окончание табл. 3.1

Атрибут	**Назначение**
*mm	Виртуальная память процесса
exit_state	Состояние завершающегося процесса
exit_code	Код возврата процесса
exit_signal	Сигнал, по которому завершился процесс
pid	Идентификатор процесса – основной атрибут процесса
tgid	Группа потоков
*real_parent	Родительский процесс
children	Потомки
sibling	Процессы с одним родителем
*group_leader	Лидер группы
comm[TASK_COMM_LEN]	Имя программы, которую выполняет процесс
*fs	Информация о файловой системе
*files	Таблица дескрипторов открытых файлов
*nsproxy	Пространство имен
*signal	Обработчики сигналов
*cgroups	Контрольные группы

У каждого процесса есть идентификатор (process ID — PID), положительное целое число, уникальным образом идентифицирующее процесс в системе. Идентификаторы процессов используются и возвращаются различными системными вызовами. PID также используется при необходимости создания идентификатора, который будет

уникальным для процесса. Идентификатор вызывающего процесса возвращается системным вызовом **getpid()**.

Ядро ограничивает максимальное количество процессов в системе, устанавливая максимально возможный PID. При создании нового процесса ему присваивается следующий по порядку PID. Всякий раз при достижении ограничения идентификаторов ядро перезапускает свой счетчик идентификаторов процессов, чтобы они назначались, начиная с наименьших целочисленных значений.

Верхний порог для идентификаторов процессов можно изменить, задав значение в файле **/proc/sys/kernel/pid_max** (которое на единицу больше, чем максимально возможное количество идентификаторов процессов). На 32-разрядной платформе максимальным значением для этого файла является 32 768, но на 64-разрядной платформе оно может быть установлено в любое значение вплоть до 2^22 (приблизительно 4 млн), позволяя справиться с очень большим количеством процессов. Это можно посмотреть следующим образом:

```
$ uname -m
x86_64
$ cat /proc/sys/kernel/pid_max
4194304
```

У каждого процесса есть процесс-родитель, который его создал с помощью системного вызова **fork()**. Определить идентификатор своего родительского процесса вызывающий процесс может с помощью системного вызова **getppid()**.

Процесс с PID==1 – это процесс **systemd**, который запускается ядром в конце процедуры начальной загрузки. Этот процесс отвечает за запуск сервисов операционной системы в пространстве пользователя и перевод системы в некоторое состояние (target), например в многопользовательский режим. Процесс **systemd** никогда не завершается. Для ядра это обычный пользовательский процесс.

С помощью PID процессы организуются в древовидную структуру с процессом **systemd** в основании дерева. Родитель каждого процесса имеет собственного родителя и т.д., возвращаясь в конечном итоге к процессу **systemd** – предку всех процессов. Вывести дерево процессов на экран можно с помощью команды **pstree**.

Если у процесса-потомка завершился родитель, то его родительским процессом становится процесс **systemd** (**getppid()** вернет 1).

Родитель любого процесса может быть найден при просмотре поля **PPid** в файле **/proc/<pid>/status**.

3.3. Жизненный цикл процесса

С помощью функции **fork()** можно создавать новые процессы, с помощью функций семейства **exec()** – запускать новые программы. Функция **exit()** и функции семейства **wait()** обслуживают процедуры выхода и ожидания завершения. Жизненный цикл процесса показан на рис. 3.1.

3.4. Создание процесса

Для создания нового процесса предназначен системный вызов **fork()**.

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

Новый процесс называется *процессом-потомком*. Эта функция вызывается один раз, а управление возвращает дважды, с единственным отличием – в процессе-потомке она возвращает 0, а в родительском процессе – идентификатор созданного процесса-потомка.

Последнее обстоятельство объясняется тем, что у процесса может быть много потомков, а в ядре нет функций, с помощью которых можно было бы получить их идентификаторы. В процессе-потомке функция **fork()** возвращает 0, поскольку процесс-потомок имеет только одного родителя и всегда может получить его идентификатор с помощью функции **getppid()**. Идентификатор процесса 0 зарезервирован за ядром, поэтому невозможно получить 0 в качестве идентификатора процесса-потомка.

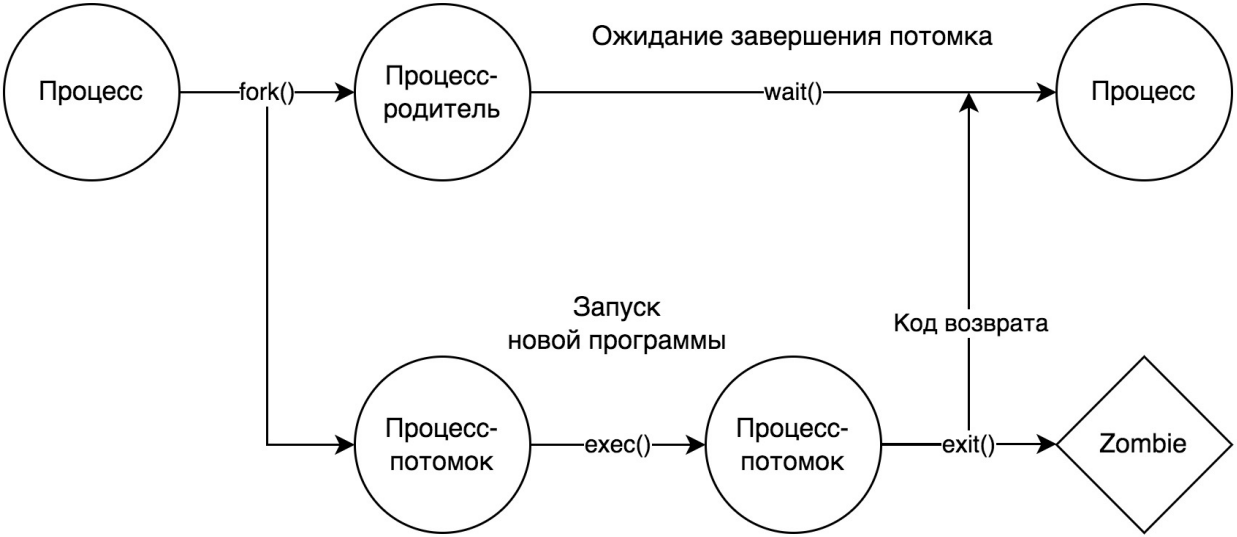


Рис. 3.1. Жизненный цикл процесса

Процесс-родитель и процесс-потомок продолжают выполнение инструкций программы, следующих за вызовом функции **fork()**. Процесс-потомок представляет собой копию родительского процесса. Таким образом, потомок получает в свое распоряжение копии сегмента данных, кучи и стека родителя. Родитель и потомок совместно используют сегмент кода. Не производится немедленное копирование сегментов данных, стека и кучи, поскольку очень часто вслед за вызовом **fork()** сразу же следует вызов **exec()**. Вместо этого используется метод, который получил название *копирование при записи* (copy-on-write, COW). Указанные выше области памяти используются совместно обоими процессами, но ядро делает их доступными только для чтения. Если один из процессов попытается изменить данные в этих областях, ядро немедленно сделает только копию конкретного участка памяти; обычно это *страница* виртуальной памяти.

Кроме того, есть возможность создания новых процессов с помощью системного вызова **clone()**. Это более универсальный вариант функции **fork()**, который позволяет вызывающему процессу определить, что будет совместно использоваться потомком и родителем.

В общем случае нельзя сказать точно, какой из двух процессов первым получит управление после вызова функции **fork()** – родитель или потомок. Это во многом зависит от алгоритма планирования, используемого ядром. Если необходимо синхронизировать работу процессов, то потребуется организовать некоторое взаимодействие между ними.

Процесс-потомок наследует от процесса-родителя следующие характеристики:

- реальный идентификатор пользователя, реальный идентификатор группы, эффективный идентификатор пользователя, эффективный идентификатор группы;
- идентификаторы дополнительных групп;
- идентификатор группы процессов;
- идентификатор сессии;
- управляющий терминал;
- текущую рабочую директорию;
- корневую директорию;
- маску режима создания файлов;

- маску сигналов и их диспозицию;
- дескрипторы открытых файлов;
- флаги **closeonexec** для открытых дескрипторов;
- среду окружения;
- присоединенные сегменты разделяемой памяти;
- отображения в память;
- ограничения на ресурсы.

Существуют следующие отличия между процессами:

- функция **fork()** возвращает различные значения;
- различные идентификаторы процессов;
- различные идентификаторы родительских процессов;
- значения таймеров в процессе-потомке устанавливаются равными 0;
- блокировки файлов, установленные в родительском процессе, не наследуются;
- таймеры, ожидающие срабатывания, в процессе-потомке сбрасываются;
- набор сигналов, ожидающих обработки, в процессе-потомке очищается.

Рассмотрим фрагмент программы **procexec/t_fork.c** [9]. Программа определяет две переменные – одну в сегменте данных и одну в стеке, и создает процесс-потомок с помощью системного вызова **fork()**. Программа демонстрирует, что процессы родитель и потомок получают отдельные копии сегментов данных и стека.

Листинг 3.1. Фрагмент программы procexec/t_fork.c.

```
/* Глобальная переменная в сегменте инициализирован-
ных данных */
static int idata = 111;
int
main(int argc, char *argv[])
{
    int istack = 222; /* Локальная переменная в стеке
*/
    pid_t childPid;

    switch (childPid = fork()) {
    case -1:
        errExit("fork");
```

```
    case 0: /* процесс-потомок */
/* Изменить переменные в сегменте данных и стеке */
        idata *= 3;
        istack *= 3;
        break;

    default: /* родительский процесс */
/* Заснуть на три секунды, чтобы выполнился потомок
*/
        sleep(3);
        break;
    }

/* Этот код выполняется и родителем и потомком */

    printf("PID=%ld %s idata=%d istack=%d\n", (long)
getpid(), (childPid == 0) ? "(child) " : "(parent)",
idata, istack);

    exit(EXIT_SUCCESS);
}
```

После запуска программы получим:

```
$ ./t_fork
PID=17092 (child)  idata=333 istack=666
PID=17091 (parent) idata=111 istack=222
```

Видно, что переменные в процессе-потомке были изменены, а в процессе-родителе остались исходные значения.

3.5. Завершение процесса

В [10] описаны восемь способов завершения работы процесса. Выделим два основных:

- нормальное завершение процесса с помощью системного вызова **_exit()**;
- аварийное завершение процесса, вызванное получением процессом сигнала с реакцией по умолчанию.

Для нормального завершения процесса предназначен системный вызов **_exit()**. Обратите внимание, что **exit()** – это библиотечная функция.

```
#include <unistd.h>

noreturn void _exit(int status);
```

Системный вызов **_exit()** немедленно завершает процесс. Все дескрипторы открытых файлов закрываются. Все потомки данного процесса переходят к процессу **systemd**. Процессу-родителю данного процесса посылается сигнал SIGCHLD.

Аргумент **status** задает код возврата процесса. Несмотря на то, что он имеет тип **int**, процессу-родителю возвращаются только младшие 8 бит (status & 0xFF) в качестве кода возврата процесса. Этот код процесс-родитель может получить с помощью семейства системных вызовов **wait()**.

Код 0 принято считать признаком успешного завершения, а любые другие значения говорят о том, что процесс завершился аварийно. Определены две константы: EXIT_SUCCESS (0) и EXIT_FAILURE (1), которые могут использоваться в качестве аргумента системного вызова **_exit()** для повышения читаемости программы.

Системный вызов **_exit()** никогда не возвращает значения.

Функция-обертка **_exit()** библиотеки языка С выполняет системный вызов **exit_group()** для того, чтобы завершить все потоки процесса. Сам системный вызов **_exit()** завершает только вызывающий поток.

3.6. Мониторинг процесса-потомка

Процесс-родитель должен отслеживать изменения состояний своих потомков – когда они завершают работу или останавливаются по сигналу. Существует два метода мониторинга процессов-потомков: семейство системных вызовов **wait()** и перехват сигнала SIGCHLD, которым ядро извещает процесс-родителя о завершении процесса-потомка.

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

pid_t wait(int *wstatus);
pid_t waitpid(pid_t pid, int *wstatus, int options);
```

В случае успеха **wait()** возвращает PID завершившегося процесса-потомка. В случае ошибки возвращает -1.

В случае успеха **waitpid()** возвращает PID процесса-потомка, у которого изменилось состояние. Если задан макрос WNOHANG, и один или более процессов-потомков, соответствующих аргументу **pid**, существуют, но еще не изменили свой статус, возвращается 0. В случае ошибки возвращается -1.

В табл. 3.2 собраны различия между системными вызовами **wait()** и **waitpid()**.

Таблица 3.2

Сравнение системных вызовов **wait()** и **waitpid()**

wait()	waitpid()
Можно отслеживать только следующего завершившегося потомка (ели потомков несколько)	Можно отслеживать конкретного потомка
Если все потомки живы, вызов блокируется до тех пор, пока какой-нибудь из потомков не завершится	Можно организовать неблокирующее ожидание, чтобы, если все потомки еще живы, немедленно узнать об этом
Можно отслеживать только завершение потомка	Можно отслеживать потомков, которые завершились, а также потомков, которые были остановлены (по сигналу SIGSTOP или SIGTTIN) или возобновили свою работу (по сигналу SIGCONT)

Процесс-потомок, который завершился, но не был ожидаем процессом-родителем, переходит в состояние «зомби» («zombie»). Ядро сохраняет в дескрипторе процесса-зомби минимальный набор информации (PID, статус завершения, статистику использования ресурсов) для того, чтобы процесс-родитель смог позже выполнить системный вызов **wait()** (или **waitpid()**) и считать эту информацию. До тех пор, пока процесс-зомби полностью не исчезнет, он будет занимать дескриптор процесса, и если в системе будет достигнуто максимальное количество процессов, то будет невозможно создать новый процесс. Как уже было сказано, максимально возможное количество процессов (и нитей) можно получить из файла **/proc/sys/kernel/pid_max**, например:

```
$ cat /proc/sys/kernel/pid_max
4194304
```

На платформе x86-64 pid_max может быть установлено в любое значение до 2^22 (PID_MAX_LIMIT, примерно 4 млн).

Если процесс-родитель завершится, порожденные им процессы-зомби унаследуются процессом **systemd**, который выполнит системный вызов **wait()**, чтобы удалить процессы-зомби.

Функции-обертки **wait()** и **waitpid()** библиотеки языка C выполняют нестандартный системный вызов **wait4()**. Для создания портируемых программ его использовать напрямую не рекомендуется.

Рассмотрим пример программы, приведенной в справочной странице **wait(2)**. В программе показано использование системных вызовов **fork()** и **waitpid()**. Программа создает процесс-потомка. Если программа запускается без аргумента, то процесс-потомок приостанавливает свое выполнение с помощью системного вызова **pause()**, чтобы пользователь смог послать ему сигнал. Если программа запускается с аргументом, то процесс-потомок немедленно завершается, передавая значение аргумента в качестве кода возврата. Процесс-родитель в цикле выполняет системный вызов **waitpid()**, чтобы отслеживать процесс-потомка, и использует макросы **W*()** для анализа значения полученного статуса.

Листинг 3.2. Программа p_wait.c.

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

int main(int argc, char *argv[])
{
    pid_t cpid, w;
    int wstatus;

    cpid = fork();
    if (cpid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }

/* Процесс-потомок */
    if (cpid == 0) {
        printf("Child PID is %jd\n", (intmax_t)
getpid());
/* Если нет аргументов, то начинаем ждать сигнала */
        if (argc == 1)
            pause();
/* Если есть аргумент, то сразу завершиться
   и значение аргумента отправить как код возврата */
        _exit(atoi(argv[1]));

/* Процесс-родитель */
    } else {
        do {
            w = waitpid(cpid, &wstatus, WUNTRACED |
WCONTINUED);
            if (w == -1) {
                perror("waitpid");
                exit(EXIT_FAILURE);
            }
            if (WIFEXITED(wstatus)) {
                printf("exited, status=%d\n", WEXIT-
STATUS(wstatus));
            } else if (WIFSIGNALED(wstatus)) {
```

```
                printf("killed by signal %d (%s)\n",
                    WTERMSIG(wstatus),        strsig-
nal(WTERMSIG(wstatus)));
            } else if (WIFSTOPPED(wstatus)) {
                printf("stopped by signal %d (%s)\n",
                    WSTOPSIG(wstatus),        strsig-
nal(WSTOPSIG(wstatus)));
            } else if (WIFCONTINUED(wstatus)) {
                printf("continued\n");
            }
        }  while  (!WIFEXITED(wstatus)  &&  !WIFSIG-
NALED(wstatus));
        exit(EXIT_SUCCESS);
    }
}
```

Запустим программу в фоновом режиме (для этого в BASH нужно добавить в конце команды символ ‘&’), и будем посылать сигналы процессу-потомку:

```
$ ./p_wait &
[1] 166674
Child PID is 166678
$ kill -STOP 166678
stopped by signal 19 (Stopped (signal))
$ kill -CONT 166678
continued
$ kill -TERM 166678
killed by signal 15 (Terminated)
[1]+  Done                    ./p_wait
```

Теперь запустим программу с числом в качестве первого аргумента:

```
$ ./p_wait 22
Child PID is 162849
exited, status=22
```

Видно, что процесс-потомок сразу завершился, передав значение аргумента в качестве кода возврата.

3.7. Запуск новой программы

Функция **fork()** часто используется для создания нового процесса, который затем запускает другую программу с помощью одной из функций семейства **exec**. Когда процесс вызывает одну из функций **exec**, то его коды полностью замещаются кодами другой программы, и эта новая программа начинает выполнение собственной функции **main()**. Идентификатор процесса при этом не меняется, поскольку функция **exec()** не создает новый процесс, она просто замещает в текущем процессе его сегмент кода, сегмент данных, динамическую область памяти и сегмент стека другой программой.

Существует шесть различных функций **exec()**, но только функция **execve()** является системным вызовом. Остальные пять – обычные библиотечные функции, которые в конечном счете обращаются к системному вызову **execve()**.

```
#include <unistd.h>

int execve(const char *pathname, char *const argv[],
                        char *const envp[]);
```

Каждый дескриптор открытого процессом файла имеет флаг **close-on-exec** (закрыть при вызове **exec**). Если этот флаг установлен, дескриптор закрывается функцией **exec**. По умолчанию после вызова функции **exec** дескриптор остается открытым, если флаг **close-on-exec** не был специально установлен с помощью функции **fcntl()**.

Обратите внимание, что реальные идентификаторы пользователя и группы не изменяются при вызове функции **exec**, но эффективные идентификаторы могут быть изменены в зависимости от состояния битов **set-user-ID** и **set-group-ID** файла запускаемой программы. Если бит **set-user-ID** установлен, то в качестве эффективного идентификатора пользователя процесса принимается идентификатор владельца файла программы. В противном случае эффективный идентификатор пользователя не изменяется. На рис. 3.2 показана последовательность запуска программы с установленным битом set-user-ID.

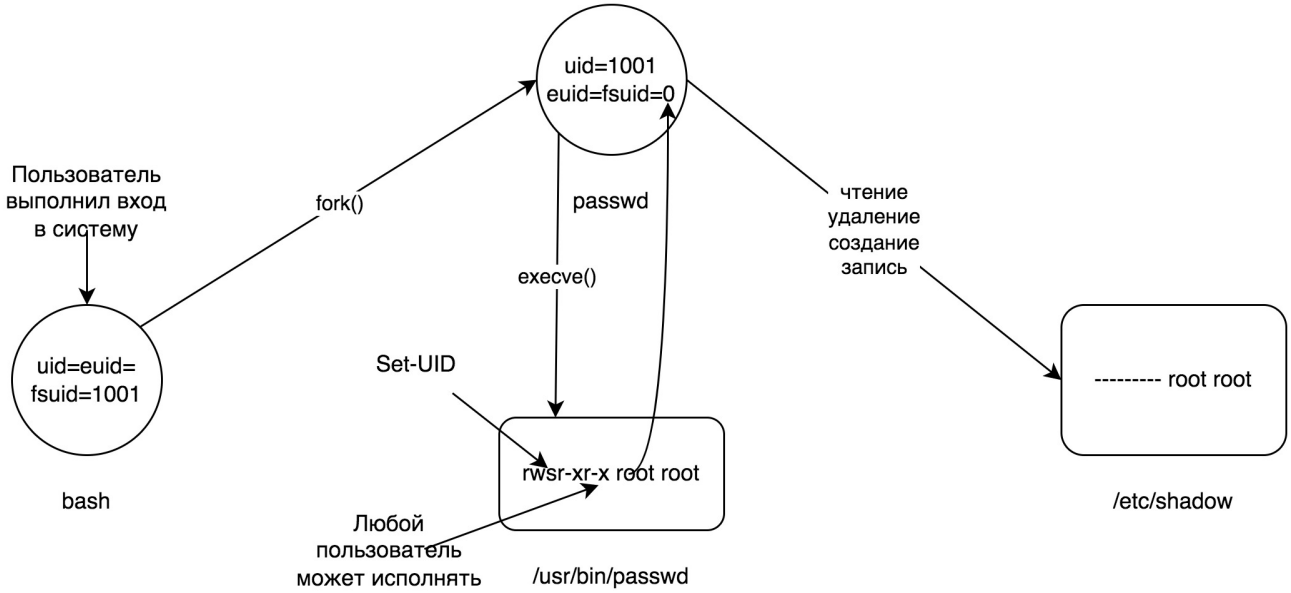


Рис. 3.2. Запуск программы с битом **set-user-ID**

Эффективный идентификатор группы устанавливается аналогичным образом.

3.8. Механизм привилегий

Механизм привилегий (capabilities) позволяет разделить неограниченные права суперпользователя (у которого EUID==0) по администрированию системы на отдельные группы прав, которые могут быть независимо друг от друга предоставлены обычным пользователям (у которых EUID≠0). Этим достигается реализация принципа *наименьших привилегий*, а также уменьшается *поверхность атаки*: если программа, обладающая одной или более привилегиями скомпрометирована, ее возможности по нанесению ущерба системе будут меньше по сравнению с такой же программой, выполняющейся с EUID==0.

Ядро Linux поддерживает около сорока привилегий. Привилегии хранятся в виде бинарных 64-битных векторов – упорядоченных наборов, где каждый бит соответствует одной определенной привилегии. Если бит установлен (равен 1), то привилегия предоставлена; если бит сброшен (равен 0), то привилегия не предоставлена.

Файл **/proc/sys/kernel/cap_last_cap** содержит числовое значение самой старшей привилегии, поддерживаемой ядром, т.е. это число показывает старший бит, который может быть установлен в векторе привилегий. Например:

```
$ uname -r
5.11.22-100.fc32.x86_64
$ cat /proc/sys/kernel/cap_last_cap
40
```

В табл. 3.3 показаны некоторые привилегии.

Привилегии можно назначать процессам и исполняемым файлам.

Таблица 3.3

Назначение некоторых привилегий

Название	Назначение
CAP_AUDIT_CONTROL	Включает и выключает аудит ядра. Изменяет правила фильтрации. Получает статус и правила фильтрации
CAP_AUDIT_READ	Читать журнал аудита через групповой netlink-сокет
CAP_AUDIT_WRITE	Писать в журнал аудита
CAP_CHOWN	Произвольно менять UID и GID файла
CAP_DAC_OVERRIDE	Права на чтение и запись любого типа файла. Право на исполнение обычного файла, если у него установлен хотя бы один из трех битов исполнения. Право на поиск в директории
CAP_DAC_READ_SEARCH	Право на чтение любого типа файла. Право на поиск в директории
CAP_FOWNER	Право выступать владельцем любого файла
CAP_LINUX_IMMUTABLE	Право устанавливать флаги инода FS_APPEND_FL и FS_IMMUTABLE_FL
CAP_SETFCAP	Запись в расширенный атрибут файла **security.capability**
CAP_SYS_ADMIN	Очень широкий набор действий. Почти все права пользователя **root**

3.9. Привилегии процесса

Каждый процесс обладает пятью 64-битными векторами привилегий. Эти векторы хранятся в структуре **cred**, на которую ссылается указатель из дескриптора процесса **task_struct**. Рассмотрим фрагмент структуры **cred**:

```
struct cred {
…
      kernel_cap_t      cap_inheritable;
      kernel_cap_t      cap_permitted;
      kernel_cap_t      cap_effective;
      kernel_cap_t      cap_bset;
      kernel_cap_t      cap_ambient;
…
}
```

Тип **kernel_cap_t** является структурой, состоящей из двух 32-битных беззнаковых целых (unsigned int):

```
typedef struct kernel_cap_struct {
      __u32 cap[_KERNEL_CAPABILITY_U32S];
} kernel_cap_t;
```

где _KERNEL_CAPABILITY_U32S = 2.

Наследуемый вектор (inheritable) – привилегии, которые сохраняются при выполнении вызова **execve()**. Наследуемые привилегии остаются наследуемыми при выполнении любой программы. Наследуемые привилегии добавляются к разрешенному набору при выполнении программы, у которой есть соответствующие биты в наследуемом наборе файла.

Разрешенный вектор (permitted) – максимальный набор привилегий, которые может использовать процесс (является надмножеством эффективного набора). Это также максимальный набор привилегий, которые могут быть добавлены в наследуемый набор процессом, не имеющим привилегию CAP_SETPCAP в своем эффективном наборе. Если процесс удаляет привилегию из своего разрешенного набора, он никогда не сможет повторно получить эту возможность.

Исключения составляют случаи запуска в процессе программы с установленным битом **set-user-ID** или программы с установленной привилегией на исполняемый файл.

Эффективный вектор (effective) – набор привилегий, используемый ядром при выполнении проверок доступа для процесса (permission checks).

Ограничивающий вектор (bounding) – механизм, который может использоваться для ограничения привилегий, получаемых во время вызова execve().

Внешний вектор (ambient) – привилегии, которые сохраняются при выполнении вызова **execve()** привилегированной программой. Привилегия может быть внешней, если она одновременно разрешенная и наследуемая. Внешние привилегии автоматически сбрасываются, если соответствующая разрешенная или наследуемая привилегия сбрасывается.

Выполнение программы, которая изменяет UID или GID из-за наличия битов **set-user-ID** или **set-group-ID**, или выполнение программы, у которой установлены какие-либо файловые привилегии, очистит внешний набор.

Внешние привилегии добавляются в разрешенный набор и присваиваются эффективному набору при вызове **execve()**.

Процесс-потомок наследует копии векторов привилегий процесса-родителя.

Посмотреть привилегии процесса можно в файле **/proc/<pid>/status**. Рассмотрим на примере обычного пользователя user1 привилегии процесса **bash**.

```
$ id
uid=1001(user1) gid=1001(user1) groups=1001(user1)
context=unconfined_u:unconfined_r:unconfined_t:s0-
s0:c0.c1023
$ cat /proc/self/status | grep ^Cap
CapInh:     0000000000000000
CapPrm:     0000000000000000
CapEff:     0000000000000000
CapBnd:     000001ffffffffff
CapAmb:     0000000000000000
$ capsh --print
```

```
Current: =
Bounding set =cap_chown,cap_dac_over-
ride,cap_dac_read_search,cap_fowner,cap_fsetid,cap_ki
ll,cap_setgid,cap_setuid,cap_setpcap,cap_linux_immu-
table,cap_net_bind_service,cap_net_broad-
cast,cap_net_ad-
min,cap_net_raw,cap_ipc_lock,cap_ipc_owner,cap_sys_mo
dule,cap_sys_rawio,cap_sys_chroot,cap_sys_ptrace,cap_
sys_pacct,cap_sys_ad-
min,cap_sys_boot,cap_sys_nice,cap_sys_re-
source,cap_sys_time,cap_sys_tty_con-
fig,cap_mknod,cap_lease,cap_audit_write,cap_au-
dit_control,cap_setfcap,cap_mac_override,cap_mac_ad-
min,cap_syslog,cap_wake_alarm,cap_block_sus-
pend,cap_audit_read,38,39,40
Ambient set =
Securebits: 00/0x0/1'b0
 secure-noroot: no (unlocked)
 secure-no-suid-fixup: no (unlocked)
 secure-keep-caps: no (unlocked)
 secure-no-ambient-raise: no (unlocked)
uid=1001(user1)
gid=1001(user1)
groups=1001(user1)
$ capsh --decode=0000000000000001
0x0000000000000001=cap_chown
```

Команда **id** выводит информацию о пользователе. Затем из файла **/proc/self/status** получаем содержимое пяти векторов привилегий процесса **bash**.

Для исследования и конструирования наборов привилегий предназначена команда **capsh**. Синтаксис команды **capsh** следующий:

```
capsh [OPTION]
```

Опция **--print** выводит привилегии процесса в мнемоническом виде и дополнительную информацию.

Опция **--decode=N** выводит название привилегии по ее номеру.

Системные вызовы **capget()** и **capset()** позволяют процессу манипулировать собственными векторами привилегий:

```
#include <linux/capability.h>
#include <sys/syscall.h>
#include <unistd.h>

int syscall(SYS_capget, cap_user_header_t hdrp,
                   cap_user_data_t datap);
int syscall(SYS_capset, cap_user_header_t hdrp,
                   const cap_user_data_t datap);
```

Библиотека **glibc** не предоставляет функций-оберток для этих системных вызовов, поэтому для их выполнения нужно использовать универсальную обертку **syscall()**. Для создания портируемых программ нужно использовать библиотечные функции **cap_set_proc()** и **cap_get_proc()**.

3.10. Привилегии файла

Привилегии также можно связать с исполняемым файлом с помощью команды **setcap**. Привилегии файла хранятся в его расширенных атрибутах в пространстве **security.capability**. Запись в этот расширенный атрибут требует наличия привилегии CAP_SETFCAP.

Суперпозиция привилегий исполняемого файла и привилегий вызывающего процесса определяет привилегии процесса после выполнения системного вызова execve().

Существует три набора привилегий файла:

- разрешенный набор (permitted) – эти привилегии автоматически разрешаются процессу, независимо от наследуемых привилегий процесса;
- наследуемый набор (inheritable) – логически перемножается с наследуемым набором процесса (операция логического «И»), чтобы определить, какие наследуемые привилегии должны быть включены в разрешенном наборе процесса после вызова **execve()**;
- эффективный бит (effective). Если этот бит установлен, то во время выполнения вызова **execve()** все новые разрешенные привилегии процесса также поднимаются в векторе эффективных привилегий. Если этот бит не установлен, то после

execve() никаких новых разрешенных привилегий нет в новом эффективном наборе.

3.11. Вычисление привилегий во время вызова execve()

Во время выполнения вызова **execve()** ядро вычисляет новые привилегии процесса по следующему алгоритму:

```
P'(ambient) = (файл привилегированный) ? 0 :
P(ambient)

P'(permitted) = (P(inheritable) & F(inheritable)) |
                                (F(permitted) & P(bound-
ing)) | P'(ambient)

P'(effective) = F(effective) ? P'(permitted) : P'(am-
bient)

P'(inheritable) = P(inheritable) [не изменяется]

P'(bounding) = P(bounding) [не изменяется]
```

где P() – привилегии процесса до вызова **execve()**; P'() – привилегии процесса после вызова **execve()**; F() – привилегии исполняемого файла.

Относительно механизма привилегий программы можно разделить на две группы:

- программы, использующие привилегии, которые скомпилированы с библиотекой **libcap** (можно посмотреть с помощью команды **ldd**);
- программы, не использующие механизм привилегий (capability-dumb).

Рассмотрим привилегии исполняемого файла на примере команды **cat**. Пусть обычный пользователь попытается вывести на экран содержимое файла **/etc/shadow**, в котором хранятся хэши паролей пользователей, и который доступен только пользователю **root**:

```
$ whereis -b cat
cat: /usr/bin/cat
$ ldd /usr/bin/cat
linux-vdso.so.1 (0x00007ffc48c74000)
libc.so.6 => /lib64/libc.so.6 (0x00007f5304859000)
/lib64/ld-linux-x86-64.so.2 (0x00007f5304a4a000)
$ cat /etc/shadow
cat: /etc/shadow: Permission denied
```

Видно, что в выводе команды **ldd** отсутствует библиотека **libcap**. Это означает, что программа **cat** не умеет управлять своими привилегиями, т.е. разработчик программы **cat** не использовал API библиотеки **libcap** для обращения к механизму привилегий.

Создадим копию программы **cat** и с помощью команды **setcap** установим для нее привилегию **cap_dac_read_search**. Так как **cat** является capability-dumb-программой, то необходимо установить оба набора привилегий (=pe) у исполняемого файла: разрешенный (p) и эффективный (e). Если бы программа **cat** умела управлять своими привилегиями, то было бы достаточно установить только разрешенный (p) набор привилегий. Команда **getcap** показывает, что привилегия установлена.

```
$ cp /usr/bin/cat .
$ sudo setcap "cap_dac_read_search=ep" cat
$ getcap ./cat
./cat = cap_dac_read_search+ep
$ ./cat /etc/shadow | head -n 1
root:!::0:99999:7:::
```

Теперь обычный пользователь может прочитать содержимое любого файла в системе, независимо от установленных для этого файла дискреционных прав доступа, т.е. может обходить механизм DAC по чтению.

Контрольные вопросы

1. Объясните суть вытесняющей многозадачности.
2. Назовите основные структуры данных, описывающих процесс.
3. Как изменить ограничение на максимально возможное количество процессов в системе?
4. Какие области памяти процесс-родитель и процесс-потомок используют совместно, а какие копируются при модификации?
5. Что происходит с открытыми файлами процесса после выполнения вызова **execve()**?
6. В какой структуре хранится маска создания файлов?
7. Какими векторами привилегий обладает процесс?
8. Где хранятся привилегии файлов?
9. В каких случаях (для каких исполняемых файлов) используется эффективный бит?
10. Какие привилегии позволяют полностью обходить дискреционный контроль доступа?

Глава 4. Адресное пространство процесса

В данной главе рассматриваются вопросы, связанные с управлением памятью процесса, и представляющие важное значение с точки зрения безопасности операционной системы. Описывается организация памяти процесса, подробно рассматривается механизм отображения файла в память процесса.

4.1. Дескриптор памяти процесса

Адресное пространство процесса состоит из всех линейных адресов, к которым процессу разрешено обращаться. Каждый процесс видит свое множество линейных адресов; адрес, используемый одним процессом, не имеет никакого отношения к адресу, используемому другим. Ядро представляет интервалы линейных адресов с помощью так называемых *областей памяти*, которые характеризуются начальным линейным адресом, длиной и правами доступа. Из соображений эффективности начальный адрес и длина области памяти должны быть кратны 4096 [11].

Для описания адресного пространства процесса ядро использует так называемый *дескриптор памяти процесса*, который представлен структурой **mm_struct**. На эту структуру указывает поле **mm** дескриптора процесса. На рис. 4.1 показан дескриптор памяти процесса, а в табл. 4.1 показаны его основные поля.

Таблица 4.1

Дескриптор памяти процесса

Атрибут	Назначение
*mmap	Указатель на голову списка областей памяти
mm_rb	Указатель на корень красно-черного дерева областей памяти
task_size	Размер виртуального пространства

Окончание табл 4.1

Атрибут	Назначение
*pgd	Указатель на глобальную директорию страниц (Page Global Directory)
map_count	Количество областей памяти. Главный счетчик использования дескриптора памяти. Каждый раз, когда значение уменьшается, ядро проверяет, достигнуто ли нулевое значение. Если значение равно нулю, дескриптор памяти уничтожается, так как больше никем не используется
start_code	Начальный адрес исполняемого кода
end_code	Конечный адрес исполняемого кода
start_data	Начальный адрес инициализированных данных
end_data	Конечный адрес инициализированных данных
start_brk	Начальный адрес кучи
brk	Текущий конечный адрес кучи
start_stack	Начальный адрес пользовательского стека
arg_start	Начальный адрес аргументов командной строки
arg_end	Конечный адрес аргументов командной строки
env_start	Начальный адрес переменных окружения
env_end	Конечный адрес переменных окружения
*exe_file	Указатель на исполняемый файл (куда указывает **/proc/<pid>/exe**)

Область памяти представлена структурой **vm_area_struct**. В табл. 4.2 показаны ее основные поля.

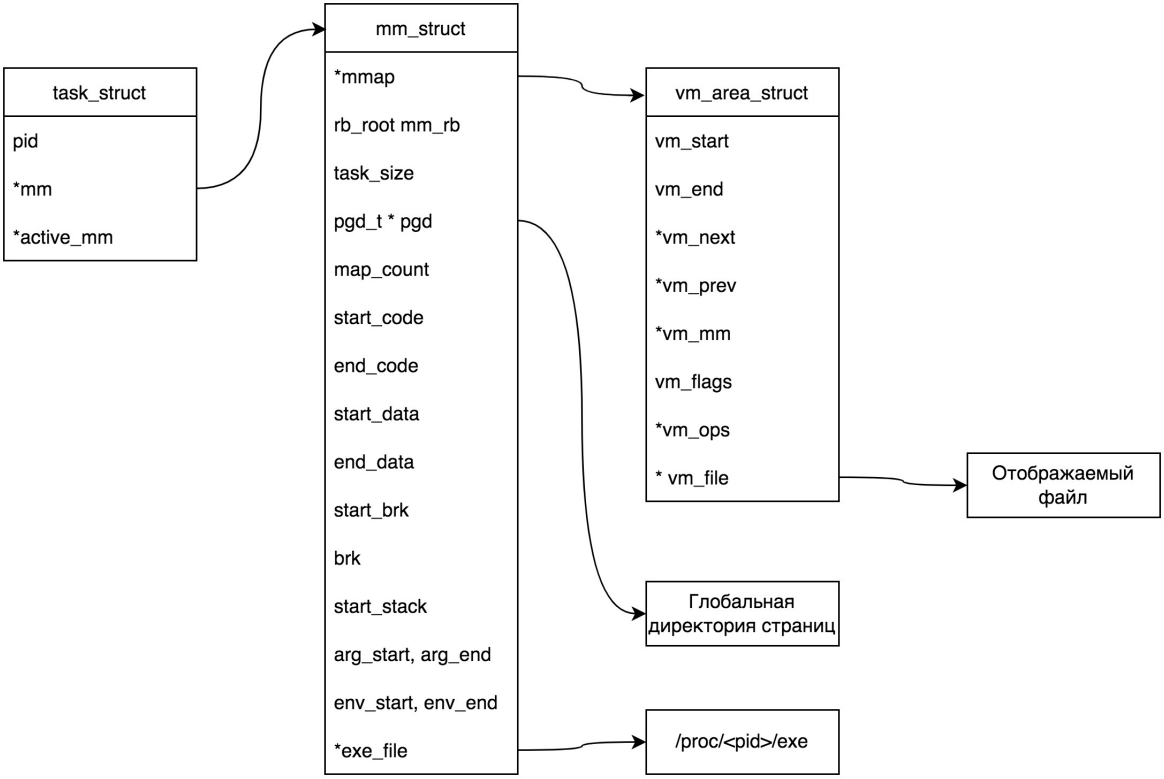


Рис. 4.1. Дескриптор памяти процесса

Таблица 4.2

Область памяти процесса

Атрибут	Назначение
vm_start	Первый линейный адрес внутри области
vm_end	Первый линейный адрес за пределами области
*vm_next *vm_prev	Связанный список областей памяти процесса, отсортированный по адресу
*vm_mm	Указатель на дескриптор памяти, к которому относится данная область
vm_page_prot	Права доступа к страницам области
vm_flags	Флаги области
*vm_ops	Указатель на операции области памяти
vm_pgoff	Смещение в отображенном файле
*vm_file	Указатель на структуру **file** отображенного файла, если такой есть

Каждый дескриптор области памяти идентифицирует интервал линейных адресов. Поле **vm_start** содержит первый линейный адрес этого интервала, а поле **vm_end** содержит первый линейный адрес за пределами интервала. Таким образом, выражение **vm_end–vm_start** определяет длину области памяти. Поле **vm_mm** указывает на дескриптор памяти процесса, владеющего данной областью памяти. Области памяти, принадлежащие процессу, никогда не перекрываются, и ядро пытается объединять области, когда новая область выделяется непосредственно рядом с существующей областью. Две смежных области могут быть объединены, если права доступа к ним совпадают [11].

4.2. Структура памяти процесса

Память процесса состоит из *сегментов*. Существует пять типов сегментов [9]:

1. *Сегмент текста* содержит двоичный код исполняемой программы, запущенной процессом. Текстовый сегмент создается только для чтения, чтобы процесс не мог случайно изменить свои собственные инструкции из-за неверного значения указателя. Поскольку многие процессы могут выполнять одну и ту же программу, текстовый сегмент создается с возможностью совместного использования. Таким образом, единственная копия кода программы может быть отображена на виртуальное адресное пространство всех процессов.

2. *Сегмент инициализированных данных* хранит глобальные и статические переменные, инициализированные явным образом. Значения этих переменных считываются из исполняемого файла при загрузке программы в память.

3. *Сегмент неинициализированных данных* содержит глобальные и статические переменные, не инициализированные явным образом. Перед запуском программы система инициализирует всю память в этом сегменте значением 0. Исторически этот сегмент часто называют «bss» (block started by symbol). Основная причина помещения прошедших инициализацию глобальных и статических переменных в отдельный от неинициализированных переменных сегмент заключается в том, что когда программа сохраняется на диске, нет никакого смысла выделять пространство под неинициализированные данные. Вместо этого исполняемой программе просто нужно записать местоположение и размер, требуемый для сегмента неинициализированных данных, и это пространство выделяется загрузчиком программы в ходе ее выполнения.

4. *Куча* – область, из которой память может динамически выделяться в ходе выполнения программы. Начало кучи называют *крайней точкой программы* (program break).

5. *Стек* содержит стековые фреймы. На платформе x86 стек располагается в верхней части памяти и растет вниз (по направлению к куче). Хотя стек растет вниз, край стека называется *вершиной*. Текущая вершина стека отслеживается в специально предназначенном

для этого регистре – указателе стека. Для каждой вызванной на данный момент функции выделяется один стековый фрейм, в котором хранятся локальные (автоматические) переменные функции, аргументы и возвращаемое значение.

Команда **size(1)** выводит размеры сегмента текста, сегментов инициализированных и неинициализированных (bss) данных исполняемой программы.

На рис. 4.2 показана структура памяти процесса на 32-разрядной платформе x86 и пример размещения переменных в разных сегментах памяти.

Область над стеком содержит аргументы командной строки программы и список переменных среды процесса. Области, закрашенные серым цветом, представляют собой недопустимые диапазоны в виртуальном адресном пространстве процесса, т.е. области, для которых не созданы таблицы страниц.

Получить аргументы командной строки можно с помощью аргумента функции **main() - char *argv[]**, который является массивом указателей на аргументы командной строки, каждый из которых представляет символьную строку, завершающуюся нулевым байтом. Первая из этих строк, **argv[0]**, содержит имя самой программы. Список указателей в **argv** завершается указателем NULL. Другим способом получения аргументов командной строки является файл **/proc/<pid>/cmdline**.

Получить список переменных среды процесса можно с помощью глобальной переменной **char **environ**, или в файле **/proc/<pid>/environ**.

Рассмотрим фрагмент программы **proc/mem_segments.c** [9], в которой продемонстрированы различные типы переменных, и в комментариях показано, в каких сегментах эти переменные размещаются. Для компиляции нужно использовать неоптимизирующий компилятор и передачу аргументов через стек.

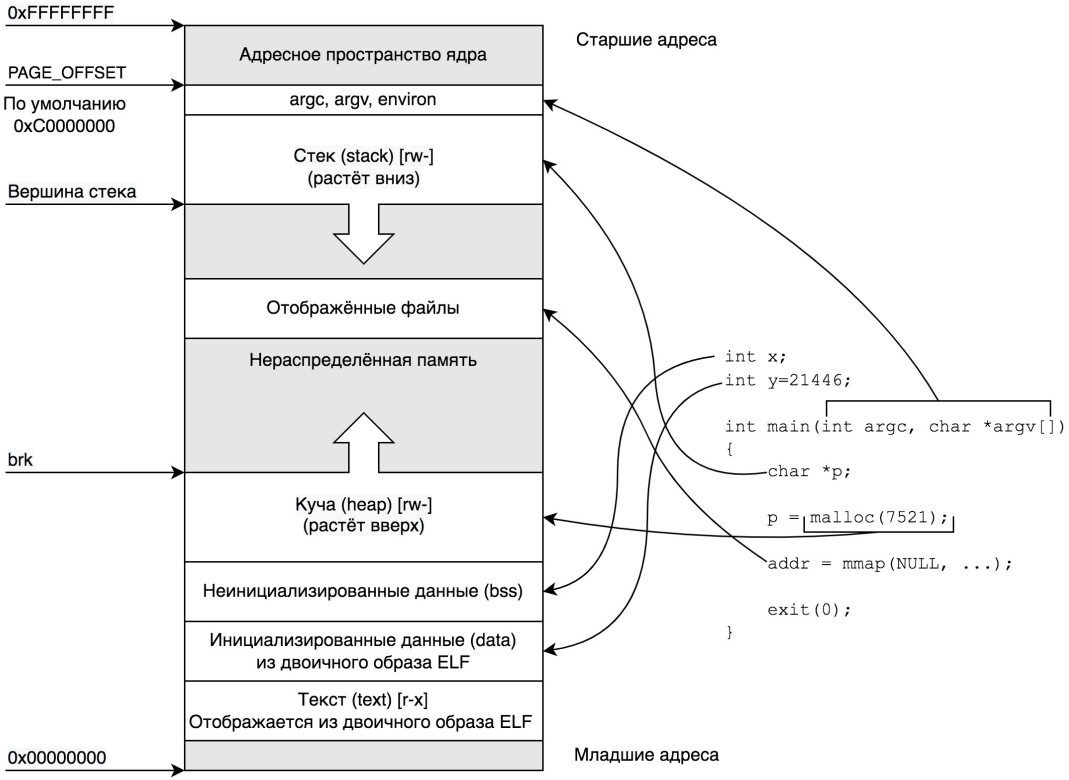


Рис. 4.2. Структура памяти процесса

Листинг 4.1. Фрагмент программы proc/mem_segments.c.

```
#include <stdio.h>
#include <stdlib.h>

char globBuf[65536];            /* Сегмент неинициали-
зированных данных */
int primes[] = { 2, 3, 5, 7 };  /* Сегмент инициализи-
рованных данных */

static int square(int x)        /* Размещается в фрейме
для square() */
{
    int result;                 /* Размещается в фрейме
для square() */

    result = x * x;
    return result;             /* Возвращаемое значение
передается через регистр */
}
static void doCalc(int val)     /* Размещается в фрейме
для doCalc() */
{
    printf("The   square   of   %d   is   %d\n",   val,
square(val));

    if (val < 1000) {
        int t;                  /* Размещается в фрейме
для doCalc() */

        t = val * val * val;
        printf("The cube of %d is %d\n", val, t);
    }
}
int
main(int argc, char *argv[])  /* Размещается в фрейме
для main() */
{
    static int key = 9973;      /* Сегмент инициализи-
рованных данных */
    static char mbuf[10240000]; /* Сегмент неинициали-
зированных данных */
```

```
    char *p;                    /* Размещается в фрейме
для main() */

    p = malloc(1024);           /* Указывает на память
в сегменте кучи */

    doCalc(key);

    exit(EXIT_SUCCESS);
}
```

В языке C есть три глобальных идентификатора: **etext**, **edata** и **end**. Они могут использоваться из программы для получения адресов следующего байта соответственно за концом сегмента текста, за концом сегмента инициализированных данных и за концом сегмента неинициализированных данных. Рассмотрим пример программы, приведенной в справочной странице **end(3)**.

Листинг 4.2. Программа p_end.c.

```
#include <stdio.h>
#include <stdlib.h>

/* Идентификаторы должны иметь тип, чтобы не было пре-
дупреждений компилятора "gcc -Wall" */
extern char etext, edata, end;

int
main(int argc, char *argv[])
{
   printf("First address past:\n");
   printf("     program text (etext)        %10p\n",
&etext);
   printf("      initialized data (edata)   %10p\n",
&edata);
   printf("      uninitialized data (end)   %10p\n",
&end);

   exit(EXIT_SUCCESS);
}
```

Скомпилируем и запустим программу:

```
$ gcc p_end.c -o p_end
$ ./p_end
First address past:
    program text (etext)         0x401235
    initialized data (edata)     0x404034
    uninitialized data (end)     0x404038
```

Структура виртуальной памяти подразумевает разбиение памяти, используемой каждой программой, на небольшие блоки фиксированного размера, называемые страницами. Соответственно, оперативная память делится на блоки страничных кадров (фреймов) одинакового размера. В любой отдельно взятый момент времени в страничных кадрах физической памяти требуется наличие только некоторых страниц программы. Эти страницы формируют так называемый резидентный набор. Копии неиспользуемых страниц программы размещаются в области подкачки – зарезервированной области дискового пространства, применяемой для дополнения оперативной памяти компьютера, – и загружаются в оперативную память лишь по мере надобности. Когда процесс ссылается на страницу, которой нет в оперативной памяти, происходит ошибка отсутствия страницы, в результате чего ядро приостанавливает выполнение процесса, пока страница загружается с диска в память [9].

В CPU за реализацию механизма виртуальной памяти отвечает блок управления памятью (Memory Management Unit, MMU). На платформе x86 размер страницы составляет 4096 байт. Программа может определить размер страницы с помощью вызова **sysconf(_SC_PAGESIZE)**. Для каждого процесса ядро поддерживает таблицу страниц. Если процесс пытается получить доступ к адресу, для которого не имеется соответствующей записи в таблице страниц, он получает сигнал SIGSEGV.

Механизм виртуальной памяти отделяет виртуальное адресное пространство процесса от физического адресного пространства оперативной памяти, что дает следующие преимущества [9]:

- изоляция процессов друг от друга и от ядра. Один процесс не может прочитать или изменить память другого процесса или ядра.

Это достигается за счет записей в таблице страниц для каждого процесса, указывающих на различные наборы физических страниц в оперативной памяти (или в области подкачки);

- совместное использование памяти несколькими процессами. Это возможно благодаря наличию записей в таблице страниц в различных процессах, ссылающихся на одни и те же страницы оперативной памяти. Совместное использование памяти происходит при двух наиболее распространенных обстоятельствах:
  - несколько процессов, выполняющих одну и ту же программу, могут совместно использовать одну и ту же (предназначенную только для чтения) копию программного кода. Эта разновидность совместного применения памяти осуществляется неявно, когда несколько программ выполняют один и тот же программный файл (или загружают одну и ту же совместно используемую библиотеку);
  - для явного запроса областей совместно используемой памяти с другими процессами процессы могут задействовать системные вызовы **shmget()** и **mmap()**. Это делается в целях обмена данными между процессами;
- упрощается реализация схем защиты памяти, т.е. записи в таблице страниц могут быть помечены, чтобы показать, что содержимое соответствующей страницы защищено от всего, кроме чтения, записи, выполнения или некоторого сочетания допустимых действий. Когда страницы оперативной памяти совместно используются несколькими процессами, можно указать, что у памяти есть защита от каждого процесса. Например, у одного процесса может быть доступ только к чтению страницы, а у другого – как к чтению, так и к записи;
- при компоновке программы не нужно знать о физическом размещении программы в оперативной памяти;
- программа загружается и запускается быстрее, поскольку в памяти требуется разместить только ее часть. Кроме того, объем памяти для среды выполнения процесса (т.е. виртуальный размер) может превышать емкость оперативной памяти.

4.3. Стековые фреймы

Стек содержит *стековые фреймы*, по одному фрейму на каждую вызванную на данный момент функцию. При вызове функции в стеке для нее создается новый фрейм, который удаляется при возврате из функции. По мере вызова функций и возврата из них стек динамически увеличивается и уменьшается.

Каждый фрейм содержит следующую информацию:

- аргументы функции;
- локальные (автоматические) переменные. Они создаются автоматически при создании фрейма и удаляются вместе с фреймом при возврате из функции;
- возвращаемое функцией значение;
- содержимое регистров. Каждая функция задействует некоторые регистры центрального процессора, например счетчик команд, указывающий на следующую инструкцию. Когда одна функция вызывает другую, копия этих регистров сохраняется в стековом фрейме вызываемой функции, чтобы при возврате из функции можно было восстановить значения соответствующих регистров для вызывающей функции.

Если функция рекурсивно вызывает саму себя, то для этой функции в стеке будет несколько фреймов.

Рассмотрим пример программы **p_stack_frame.c**, в которой определены три функции. Выполнение начинается с функции **main()**, которая вызывает функцию **f1()**, которая в свою очередь вызывает функцию **f2()**. В каждой функции определены аргументы и локальные переменные, которые каждая функция выводит на экран.

Листинг 4.3. Программа p_stack_frame.c.

```
#include <stdio.h>
#include <stdlib.h>

void f2(int b)
{
    int n=20;

    printf("f2(): b = %d, n = %d\n", b, n);
}
```

```
void f1(int a)
{
    int j=10;

    printf("f1(): a = %d, j = %d\n", a, j);

    f2(j);
}

int main(int argc, char *argv[])
{
    int i=2022;

    printf("main(): i = %d\n", i);

    f1(i);

    exit(EXIT_SUCCESS);
}
```

Скомпилируем программу, включив отладочную информацию и отключив оптимизацию, и запустим программу:

```
$ gcc -g -O0 -fno-omit-frame-pointer p_stack_frame.c -
o p_stack_frame
$ ./p_stack_frame
main(): i = 2022
f1(): a = 2022, j = 111
f2(): b = 111, n = 222
```

Теперь запустим программу под управлением интерактивного отладчика **gdb**, который обладает широкими возможностями и собственным набором подкоманд. Опция **-q** запускает отладчик без вывода справочной информации.

Установим точку останова на выходе из функции **f2()** (строка 9), запустим программу (команда **run**) и выведем список стековых фреймов (команда **backtrace**).

```
$ gdb -q ./p_stack_frame
Reading symbols from ./p_stack_frame...
(gdb) list
```

```
8         printf("f2(): b = %d, n = %d\n", b, n);
9     }
10
11    void f1(int a)
12    {
13        int j=111;
14
15        printf("f1(): a = %d, j = %d\n", a, j);
16
17        f2(j);
(gdb) break 9
Breakpoint 1 at 0x40115f: file p_stack_frame.c, line
9.
(gdb) run
Starting program: /home/user1/p_stack_frame
Missing separate debuginfos, use: dnf debuginfo-in-
stall glibc-2.31-6.fc32.x86_64
main(): i = 2022
f1(): a = 2022, j = 111
f2(): b = 111, n = 222

Breakpoint 1, f2 (b=111) at p_stack_frame.c:9
9     }
(gdb) backtrace
#0  f2 (b=111) at p_stack_frame.c:9
#1       0x0000000000401195   in   f1   (a=2022)   at
p_stack_frame.c:17
#2        0x00000000004011cc    in    main    (argc=1,
argv=0x7fffffffdff8) at p_stack_frame.c:26
(gdb)
```

Видно, что в стеке находится три фрейма: самый последний по времени под номером 0 для функции **f2()**, предыдущий под номером 1 для функции **f1()** и самый первый под номером 2 для функции **main()**.

Команда отладчика **info frame** позволяет вывести содержимое каждого фрейма:

```
(gdb) info frame 0
Stack frame at 0x7fffffffdeb0:
```

```
 rip = 0x40115f in f2 (p_stack_frame.c:9); saved rip =
0x401195
 called by frame at 0x7fffffffdee0
 source language c.
 Arglist at 0x7fffffffdea0, args: b=111
 Locals at 0x7fffffffdea0, Previous frame's sp is
0x7fffffffdeb0
 Saved registers:
  rbp at 0x7fffffffdea0, rip at 0x7fffffffdea8
(gdb) info frame 1
Stack frame at 0x7fffffffdee0:
 rip = 0x401195 in f1 (p_stack_frame.c:17); saved rip
= 0x4011cc
 called by frame at 0x7fffffffdf10, caller of frame at
0x7fffffffdeb0
 source language c.
 Arglist at 0x7fffffffded0, args: a=2022
 Locals at 0x7fffffffded0, Previous frame's sp is
0x7fffffffdee0
 Saved registers:
  rbp at 0x7fffffffded0, rip at 0x7fffffffded8
(gdb) info frame 2
Stack frame at 0x7fffffffdf10:
 rip = 0x4011cc in main (p_stack_frame.c:26); saved
rip = 0x7ffff7e0b082
 caller of frame at 0x7fffffffdee0
 source language c.
 Arglist at 0x7fffffffdf00, args: argc=1,
argv=0x7fffffffdff8
 Locals at 0x7fffffffdf00, Previous frame's sp is
0x7fffffffdf10
 Saved registers:
  rbp at 0x7fffffffdf00, rip at 0x7fffffffdf08
(gdb)
```

Команда отладчика **info locals** позволяет вывести содержимое локальных переменных функции, а команда **info arg**s – аргументы. Команды выполняются в фрейме, в котором сейчас находится отладчик. Значения переменных можно выводить на экран (команда

print) и даже менять (команда **set**). Для переключения между фреймами используется команда **frame**. Программа остановилась в функции **f2()**, поэтому отладчик находится сейчас в фрейме 0.

```
(gdb) backtrace
#0  f2 (b=111) at p_stack_frame.c:9
#1       0x0000000000401195   in   f1   (a=2022)   at
p_stack_frame.c:17
#2        0x00000000004011cc    in    main    (argc=1,
argv=0x7fffffffdff8) at p_stack_frame.c:26
(gdb) info locals
n = 222
(gdb) info args
b = 111
(gdb) print n
$7 = 222
(gdb) frame 1
#1       0x0000000000401195   in   f1   (a=2022)   at
p_stack_frame.c:17
17        f2(j);
(gdb) info locals
j = 111
(gdb) info args
a = 2022
(gdb) frame 2
#2        0x00000000004011cc    in    main    (argc=1,
argv=0x7fffffffdff8) at p_stack_frame.c:26
26        f1(i);
(gdb) info locals
i = 2022
(gdb) info args
argc = 1
argv = 0x7fffffffdff8
(gdb)
```

Теперь с помощью команды отладчика **next** выполним следующую строку программы. Функция **f2()** завершится и управление вернется в функцию **f1()**. Фрейм для функции **f2()** будет удален. Таким же способом завершим функцию **f1()** и вернемся в функцию **main()**. Фрейм для функции **f1()** будет удален. После завершения программы в стеке ничего не останется.

```
(gdb) backtrace
#0  f2 (b=111) at p_stack_frame.c:9
#1       0x0000000000401195    in    f1    (a=2022)    at
p_stack_frame.c:17
#2        0x00000000004011cc    in    main    (argc=1,
argv=0x7fffffffdff8) at p_stack_frame.c:26
(gdb) next
f1 (a=2022) at p_stack_frame.c:18
18    }
(gdb) backtrace
#0  f1 (a=2022) at p_stack_frame.c:18
#1        0x00000000004011cc    in    main    (argc=1,
argv=0x7fffffffdff8) at p_stack_frame.c:26
(gdb) next
main       (argc=1,       argv=0x7fffffffdff8)       at
p_stack_frame.c:28
28        exit(EXIT_SUCCESS);
(gdb) backtrace
#0       main    (argc=1,    argv=0x7fffffffdff8)    at
p_stack_frame.c:28
(gdb) next
[Inferior 1 (process 24235) exited normally]
(gdb) backtrace
No stack.
(gdb)
```

Таким образом, видно, что фреймы создаются и удаляются по принципу LIFO (последний пришел – первый ушел). Вывести содержимое стека можно командой отладчика **x/10xg $rsp**, где x – вывести (от eXamine), 10 – количество, x – в 16-ричном формате, g – 8-байтовые слова, rsp – регистр указателя стека, относительно значения которого будет происходить вывод 10 слов.

4.4. Отображение файлов в память процесса

Отображение файла в память процесса является важным механизмом с точки зрения информационной безопасности, так как оно используется для организации межпроцессного взаимодействия и ряда других задач. Существует два вида отображений:

1. Файловое отображение (файл, отображенный в память). Файл отображается непосредственно на виртуальную память вызывающего процесса. Содержимое файла становится доступным для байтовых операций на соответствующем участке памяти процесса. Страницы отображения по мере необходимости (автоматически) загружаются из соответствующего файла.

2. Анонимное отображение. С отображением не связан никакой файл, а его страницы заполняются нулями. Анонимное отображение можно представить в виде отображения виртуального файла, который всегда содержит нули.

Отображение можно разделять между процессами (т.е. записи таблицы со страницами каждого из процессов будут указывать на одни и те же страницы физической памяти) в двух случаях:

1) когда два процесса отображают один и тот же участок файла, страницы памяти, к которым они получают доступ, становятся для них общими;

2) процесс-потомок, созданный с помощью вызова **fork()**, наследует копии отображений процесса-родителя, указывающих на те же страницы физической памяти, что и отображения родителя.

Отображение может быть разделяемым или приватным:

Разделяемое отображение (MAP_SHARED). Изменения, вносимые в содержимое отображения, доступны другим процессам, которые разделяют то же отображение; в случае с файловым отображением изменения передаются в исходный файл.

Приватное отображение (MAP_PRIVATE). Изменения, вносимые в содержимое отображения, остаются невидимыми для других процессов; в случае с файловым отображением изменения не передаются в исходный файл. Страницы приватного отображения изначально являются разделяемыми, однако изменения каждого отображения распространяются только на процесс, которому они принадлежат. Чтобы достичь этого, ядро использует методику копирования при записи; т.е. когда процесс пытается изменить содержимое страницы, ядро создает для него ее копию (и корректирует таблицу страниц процесса).

Таким образом, возможны четыре разных варианта отображения, представленных в табл. 4.3.

Таблица 4.3

Виды отображения файла в память процесса

Видимость изменений	Назначение отображения	
	Файловое	Анонимное (MAP_ANONYMOUS)
Разделяемое (MAP_SHARED)	Ввод/вывод, отображенный в память. Разделение памяти между неродственными процессами (IPC)	Разделение памяти между родственными процессами (IPC)
Приватное (MAP_PRIVATE)	Инициализация памяти процесса из содержимого файла	Выделение памяти

Рассмотрим назначение каждого из четырех типов отображения.

При разделяемом файловом отображении все процессы, отображающие один участок одного и того же файла, разделяют одни и те же страницы физической памяти, которые изначально инициализированы с помощью этого участка. Изменения, вносимые в отображение, передаются обратно в файл. Данное отображение позволяет:

- отобразить в память ввод/вывод. Это значит, что файл загружается на какой-то участок виртуальной памяти процесса, и все изменения, вносимые в данную память, автоматически записываются в сам файл. Таким образом, ввод/вывод, отображенный в память, является альтернативой вызовам **read()** и **write()**, которые используются для чтения и записи файла;
- неродственным процессам разделять один и тот же участок памяти для обеспечения межпроцессного взаимодействия.

При разделяемом анонимном отображении каждый вызов **mmap()** создает новое отображение со своими отдельными страницами памяти. К страницам отображения не применяется процедура копирования при записи, т.е. когда потомок после вызова **fork()** наследует отображение родителя, оба процесса разделяют одни и те же страницы физической памяти, и изменения, вносимые одним

процессом, видны другому. Разделяемые анонимные отображения позволяют организовать межпроцессное взаимодействие, но только между родственными процессами.

При приватном файловом отображении содержимое отображения инициализируется с помощью участка файла. Несколько процессов, отображающих один файл, изначально разделяют одни и те же страницы памяти, но, поскольку задействовано копирование при записи, изменения, вносимые процессом в отображение, не видны остальным процессам. Основное применение этого вида отображения заключается в инициализации участка памяти содержимым файла. Например, можно инициализировать сегменты текста и данных процесса, заполнив их соответствующими участками двоичного исполняемого файла или разделяемой библиотеки.

При приватном анонимном отображении каждое отображение, создаваемое с помощью вызова **mmap()**, отличается от других отображений данного вида, созданных тем же (или другим) процессом (страницы физической памяти разные). И хотя потомок наследует отображения родителя, процедура копирования при записи гарантирует, что после вызова **fork()** родитель и потомок не будут видеть изменения друг друга, вносимые в это отображение. Основное применение приватного анонимного отображения заключается в выделении (и заполнении нулями) памяти для процесса (например, при выделении больших блоков памяти **malloc()** использует вызов **mmap()**).

Отображения, включая тип (MAP_SHARED или MAP_PRIVATE), наследуются потомками при вызове **fork()**, но теряются, когда процесс выполняет вызов **exec()**.

Информация об отображениях процесса доступна через файловую систему **/proc** в файлах **/proc/<pid>/maps**, **/proc/<pid>/smaps** директории **/proc/<pid>/map_files**.

Для работы с отображением файла предназначены системные вызовы **mmap()** и **munmap()**. Системный вызов **mmap()** создает новое отображение в виртуальном адресном пространстве вызывающего процесса. Системный вызов **munmap()** удаляет отображение из виртуального адресного пространства вызывающего процесса:

```
#include <sys/mman.h>

void *mmap(void *addr, size_t length, int prot, int
flags,
                   int fd, off_t offset);

int munmap(void *addr, size_t length);
```

Аргумент **addr** задает виртуальный адрес, по которому будет находиться отображение. Если присвоить ему значение NULL, то ядро само выберет подходящий адрес. Это предпочтительный способ создания отображения. Но также можно указать ненулевой адрес, который будет учитываться ядром при размещении отображения в памяти. На практике данный адрес будет как минимум округлен, чтобы совпасть с началом ближайшей страницы. В любом случае ядро выберет адрес, не конфликтующий ни с одним другим отображением. В случае успеха **mmap()** возвращает начальный адрес нового отображения. В случае ошибки возвращается значение MAP_FAILED. В Linux (как и в большинстве других разновидностей UNIX) константа MAP_FAILED равна ((void *) – 1).

Аргумент **length** задает размер отображения в байтах. И хотя он не должен быть кратным размеру страниц памяти в системе (возвращаемому вызовом **sysconf**(_SC_PAGESIZE)), при создании отображения ядро использует единицы измерения, благодаря которым **length** автоматически округляется до ближайшего следующего числа, кратного размеру страницы.

Аргумент **prot** задает защиту памяти нового отображения и не должен конфликтовать с режимом открытия файла. Он может быть равен либо значению PROT_NONE, либо комбинации (логическому «ИЛИ») из следующих флагов: PROT_READ, PROT_WRITE, PROT_EXEC. Значения флагов защиты представлены в табл. 4.2. Если процесс, пытаясь получить доступ к участку памяти, нарушит его защиту, то ядро пошлет этому процессу сигнал SIGSEGV.

Аргумент **flags** определяет, будут ли изменения содержимого отображения видны другим процессам, отображающим тот же регион, и будут ли изменения записаны в отображаемый файл на диске. Значения обязательных флагов и примеры дополнительных флагов представлены в табл. 4.3. Обязательные флаги являются взаимоисключающими, т.е. нужно указать один из трех:

MAP_SHARED, MAP_SHARED_VALIDATE или MAP_PRIVATE. Флаги перечисляются через логическое «ИЛИ».

Оставшиеся аргументы, **fd** и **offset**, используются в сочетании с файловыми отображениями (и игнорируются анонимными). Аргумент **fd** – это файловый дескриптор файла, который нужно отобразить. Аргумент **offset** задает начальную позицию отображения в файле и должен быть кратным размеру страницы памяти в системе. Чтобы отобразить весь файл целиком, в качестве **offset** можно указать 0, а **length** передать размер файла.

Таблица 4.4

Флаги защиты памяти отображения файла

Флаг защиты	Назначение
PROT_EXEC	Содержимое страниц можно выполнить
PROT_READ	Содержимое страниц можно прочитать
PROT_WRITE	Содержимое страниц можно изменить
PROT_NONE	Доступ к страницам запрещен

Таблица 4.5

Флаги отображения файла

Флаг	Назначение
Обязательные взаимоисключающие флаги	
MAP_SHARED	Создает разделяемое отображение. Изменения, вносимые в страницы, видны другим процессам, отображающим тот же участок с помощью атрибута MAP_SHARED. В случае с файловым отображением изменения передаются обратно в исходный файл. Обновление файла может происходить не сразу. Чтобы записать содержимое отображения обратно в исходный файл, перед удалением следует сделать вызов **msync()**

Окончание табл. 4.5

Флаг	Назначение
MAP_SHARED_ VALIDATE	Аналогично MAP_SHARED, но с проверкой всех дополнительных флагов на корректность
MAP_PRIVATE	Создает приватное отображение. Изменения, вносимые в страницы, не видны другим процессам, использующим то же отображение. В случае с файловым отображением изменения не передаются обратно в исходный файл
Дополнительные флаги	
MAP_ANONYMOUS	Отображение инициализируется нулями. Изменения не передаются ни в какой файл. Аргумент **fd** игнорируется. Аргумент **offset** должен быть равен нулю
MAP_FIXED	Аргумент **addr** интерпретируется как точный адрес, по которому нужно разместить отображение

Сведения о защите хранятся в таблицах виртуальной памяти, выделяемых для каждого отдельного процесса. Таким образом, разные процессы могут отобразить данные на один и тот же участок памяти, но с разной защитой.

Рассмотрим фрагмент программы **mmap/t_mmap.c** [9], демонстрирующей использование вызова **mmap()** для создания разделяемого файлового отображения. Сначала данная программа отображает файл, заданный в виде первого аргумента командной строки. Затем она выводит значение строки, находящейся в начале отображенного участка. Затем второй аргумент командной строки копируется на соответствующий участок разделяемой памяти.

Листинг 4.3. Фрагмент программы mmap/t_mmap.c.

```
#include <sys/mman.h>
#include <fcntl.h>
#include "tlpi_hdr.h"
```

```
#define MEM_SIZE 10

int
main(int argc, char *argv[])
{
    char *addr;
    int fd;

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s file [new-value]\n", argv[0]);

/* Открываем файл, переданный через первый аргумент, в
режиме чтения-записи */
    fd = open(argv[1], O_RDWR);
    if (fd == -1)
        errExit("open");

    addr = mmap(NULL, MEM_SIZE, PROT_READ |
PROT_WRITE, MAP_SHARED, fd, 0);
    if (addr == MAP_FAILED)
        errExit("mmap");

/* Дескриптор 'fd' больше не нужен */
    if (close(fd) == -1)
        errExit("close");

/* Выводим строку по адресу отображения. Для безопас-
ности ограничиваем вывод MEM_SIZE байтами */
    printf("Current string=%.*s\n", MEM_SIZE, addr);

/* Проверяем длину строки, переданной через второй ар-
гумент */
    if (argc > 2) {
        if (strlen(argv[2]) >= MEM_SIZE)
            cmdLineErr("'new-value' too large\n");

/* Заполняем участок памяти нулями */
        memset(addr, 0, MEM_SIZE);

/* Записываем строку, переданную через второй аргумент
*/
        strncpy(addr, argv[2], MEM_SIZE - 1);
```

```
        if (msync(addr, MEM_SIZE, MS_SYNC) == -1)
            errExit("msync");

        printf("Copied \"%s\" to shared memory\n",
argv[2]);
    }

    exit(EXIT_SUCCESS);
}
```

Откроем два терминала. В первом терминале создадим тестовый файл, заполнив его нулями. Затем выведем содержимое файла по восемь символов в строке. Обратите внимание, что команда **echo** добавляет в конец файла символ перевода строки (\n). Затем запустим программу, передав ей имя файла и строку для изменения его содержимого.

```
$ dd if=/dev/zero of=map_file.txt bs=1 count=1024
$ echo Hello world > map_file.txt
$ od -c -w8 map_file.txt
0000000   H   e   l   l   o       w   o
0000010   r   l   d  \n
0000014
$ ./t_mmap map_file.txt 'Good bye'
Current string=Hello world
Copied "Good bye" to shared memory
$ od -c -w8 map_file.txt
0000000   G   o   o   d       b   y   e
0000010  \0  \0  \0  \0
0000014
```

Во втором терминале находим процесс и с помощью файловой системы **/proc** исследуем отображение до и после внесения изменений в участок памяти.

```
# ps a
    PID TTY      STAT   TIME COMMAND
...
  55885 pts/0    S+     0:00 ./t_mmap map_file.txt Good
bye
# cat /proc/55885/maps
```

```
00400000-00401000    r--p    00000000    103:05    2938
/.../t_mmap
...
7fc388842000-7fc388843000  rw-s  00000000  103:05  2935
/.../map_file.txt
# cat /proc/55885/map_files/7fc388842000-7fc388843000
Hello world
# cat /proc/55885/map_files/7fc388842000-7fc388843000
Good bye$
```

Видно, что в памяти процесса есть несколько регионов. Регион 00400000-00401000 содержит сегмент текста программы **t_mmap**. Регион 7fc388842000-7fc388843000 содержит отображенный файл **map_file.txt**. В директории **/proc/55885/map_files/** можно обратиться к содержимому регионов, как к обычным файлам.

Рассмотрим пример программы **p_mmap**, приведенный в справочной странице **mmap(2)**. Программа выводит фрагмент файла, путь к которому передается через первый аргумент командной строки, на стандартный вывод. Границы фрагмента задаются в байтах через смещение от начала файла и длину во втором и третьем аргументах соответственно. Программа создает отображение требуемых страниц файла и затем с помощью системного вызова **write()** выводит фрагмент на экран.

Листинг 4.4. Программа p_mmap.

```
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define handle_error(msg) \
    do { perror(msg); exit(EXIT_FAILURE); } while (0)

int
main(int argc, char *argv[])
{
    char *addr;
    int fd;
    struct stat sb;
```

```
    off_t offset, pa_offset;
    size_t length;
    ssize_t s;

/* Должно быть два или три аргумента: имя файла, сме-
щение от начала, и необязательно длина */
    if (argc < 3 || argc > 4) {
       fprintf(stderr, "%s file offset [length]\n",
argv[0]);
       exit(EXIT_FAILURE);
    }

/* Открываем файл */
    fd = open(argv[1], O_RDONLY);
    if (fd == -1)
       handle_error("open");

/* Получаем размер файла */
    if (fstat(fd, &sb) == -1)
       handle_error("fstat");

/* Преобразуем смещение в число */
    offset = atoi(argv[2]);
/* Смещение для mmap() должно быть выровнено по границе
страницы */
    pa_offset = offset & ~(sysconf(_SC_PAGE_SIZE) -
1);

    if (offset >= sb.st_size) {
       fprintf(stderr, "offset is past end of
file\n");
       exit(EXIT_FAILURE);
    }

/* Преобразуем длину в число */
    if (argc == 4) {
       length = atoi(argv[3]);
/* Если длина фрагмента выходит за границы файла, то
выводим до конца файла */
       if (offset + length > sb.st_size)
         length = sb.st_size - offset;
```

```
/* Если длина не задана, то выводим до конца файла */
    } else {
       length = sb.st_size - offset;
    }

/* Создаем отображение */
    addr = mmap(NULL, length + offset - pa_offset,
PROT_READ,
             MAP_PRIVATE, fd, pa_offset);
    if (addr == MAP_FAILED)
       handle_error("mmap");

/* Выводим на экран */
    s = write(STDOUT_FILENO, addr + offset - pa_offset,
length);
    if (s != length) {
       if (s == -1)
         handle_error("write");

       fprintf(stderr, "partial write");
       exit(EXIT_FAILURE);
    }

/* Удаляем отображение */
    munmap(addr, length + offset - pa_offset);
/* Закрываем файл */
    close(fd);

    exit(EXIT_SUCCESS);
}
```

Скомпилируем и запустим программу:

```
$ gcc p_mmap.c -o p_mmap
$ ./p_mmap /etc/yum.repos.d/fedora.repo 0 9
[fedora]
```

Для примера мы вывели первые девять байтов файла с описанием репозитория дистрибутива Fedora.

Контрольные вопросы

1. Из каких сегментов состоит память процесса?
2. Какой сегмент памяти процесса не должен быть доступен на запись?
3. В каком сегменте памяти процесса размещаются глобальные переменные?
4. Что называется вершиной стека?
5. Как получить список переменных среды процесса?
6. Как получить адрес конца сегмента текста?
7. Какие преимущества дает механизм виртуальной памяти?
8. Какие существуют виды отображений файла?
9. Какое отображение может использоваться в качестве средства межпроцессного взаимодействия только между родственными процессами?
10. Почему в выводе программы (см. листинг 4.3) приглашение BASH не появилось на новой строке?

Глава 5. Система мандатного управления SELinux

В данной главе рассматривается механизм мандатного управления доступом SELinux, который существенно расширяет возможности по контролю информационных потоков в операционной системе и предотвращению атак «нулевого дня».

5.1. Классы объектов

Отличительной чертой SELinux [17–19] является гибкая реализация мандатного управления доступом. Гибкость проявляется в том, что SELinux учитывает особенности тех или иных системных ресурсов (объектов), что позволяет предоставлять права процессам очень точечно или с высокой гранулярностью. Этим достигается реализация одного из главных принципов информационной безопасности – принципа наименьших привилегий (principle of least privilege).

Перед разработчиками политики безопасности стоит ряд инженерных вопросов. Сколько нужно различать отдельных видов объектов? Сколько нужно контролировать видов доступа к этим объектам?

Например, когда процесс открывает файл, ядро считывает файл из файловой системы на диске и создает структуру данных **file** в оперативной памяти, через которую происходит работа с файлом. В SELinux структуре ядра **file** соответствует класс объектов **file**. Процесс, открывая файл, может указать в системном вызове **open()** режим работы с файлом. Для каждого режима SELinux поддерживает соответствующее право доступа. Таким образом, версия политики безопасности SELinux должна соответствовать версии ядра Linux. Если в ядре Linux появились новые объекты (т.е. новые структуры данных), то и в политике безопасности SELinux должны появиться соответствующие классы объектов с соответствующим набором прав доступа, чтобы SELinux смог контролировать доступ к этим новым объектам.

Для вывода статистической информации о политике безопасности SELinux, а также детальной информации о классах объектов, правах, TE-типах, буленах, ролях и т.д. предназначена команда **seinfo**. Формат команды следующий:

```
seinfo [опции] [выражение] [файл_политики]
```

Если файл, содержащий бинарную политику, не задан, то команда **seinfo** выводит информацию о текущей загруженной политике. Рассмотрим пример:

```
$ seinfo
Statistics for policy file: /sys/fs/selinux/policy
Policy Version:             32 (MLS enabled)
Target Policy:              selinux
Handle unknown classes:     allow
Classes:              131   Permissions:          439
Sensitivities:          1   Categories:          1024
Types:               4987   Attributes:           254
Users:                  8   Roles:                 14
Booleans:             337   Cond. Expr.:          372
Allow:              62363   Neverallow:             0
Auditallow:           168   Dontaudit:           8446
Type_trans:        254921   Type_change:           87
Type_member:           35   Range_trans:         6102
Role allow:            37   Role_trans:           430
Constraints:           72   Validatetrans:          0
MLS Constrain:         72   MLS Val. Tran:          0
Permissives:            0   Polcap:                 5
Defaults:               7   Typebounds:             0
Allowxperm:             0   Neverallowxperm:        0
Auditallowxperm:        0   Dontauditxperm:         0
Ibendportcon:           0   Ibpkeycon:              0
Initial SIDs:          27   Fs_use:                33
Genfscon:             106   Portcon:              642
Netifcon:               0   Nodecon:                0
```

В политике **targeted** версии 32 поддерживается 131 класс объектов доступа и 439 различных прав доступа. Это означает, что когда политика **targeted** загружена в SELinux, есть возможность различать 131 вид сущностей системы и к каждому виду определить специфический для него набор прав доступа, всего 439 различных прав доступа. Таким образом, по сравнению с классической ОС UNIX, достигается очень высокая гранулярность управления доступом.

Для каждого типа файла Linux в политике **targeted** определен отдельный класс объектов:

```
$ seinfo --class | grep -E 'file|dir'
blk_file
chr_file
dir
fifo_file
file
filesystem
lnk_file
sock_file
```

Дополнительно определен отдельный класс объектов для самой файловой системы, которая с точки зрения SELinux также является объектом доступа со своими специфическими свойствами в отличие от Linux, где доступ к файловой системе контролируется через блочное устройство, содержащее файловую систему, а также предоставлением прав монтировать, перемонтировать, размонтировать или управлять квотами на файловой системе только пользователю **root**.

В этой главе часто сравниваются стандартные права доступа Linux, которые рассматривались в предыдущих главах, с правами SELinux. Для более понятного изложения будем называть стандартные права дискреционного управления доступом ***правами DAC***. Тогда, права доступа, поддерживаемые SELinux, можно условно разделить на три группы:

- права DAC (rwx);
- дополнительные права Linux, расширяющие стандартную модель DAC;
- права SELinux, которые имеют смысл только с загруженной политикой безопасности.

Исследуем классы объектов на примере трех фундаментальных сущностей системы и соответствующих им прав доступа. Сначала рассмотрим классы **file** и **dir**, которые соответствуют обычному файлу и директории, а затем класс **process**, который соответствует процессу:

```
$ seinfo --class file -x

Classes: 1
class file
inherits file
{
      execute_no_trans
      entrypoint
}
```

Видно, что класс **file** обладает общим набором прав **file**, а также двумя специфическими для SELinux правами **execute_no_trans** и **entrypoint**.

```
$ seinfo --class dir -x

Classes: 1
class dir
inherits file
{
      reparent
      search
      remove_name
      add_name
      rmdir
}
```

Видно, что класс **dir** обладает общим набором прав **file**, а также пятью специфическими для директорий правами доступа.

Рассмотрим общий набор прав **file**, который отражает операции, выполняемые как над обычным файлом, так и над директорией:

```
$ seinfo --common file -x

Commons: 1
common file
{
      swapon
      getattr
      unlink
      write
```

```
        rename
        lock
        open
        ioctl
        execute
        relabelto
        audit_access
        link
        append
        mounton
        relabelfrom
        map
        quotaon
        execmod
        read
        setattr
        create
}
```

Рассмотрим класс **process**. Особенностью процессов является то, что процесс может выступать как в роли субъекта, так и в роли объекта доступа. Например, если один процесс посылает сигнал другому процессу, то посылающий процесс является субъектом, а процесс, принимающий сигнал, – объектом доступа. Процесс обладает рядом атрибутов, для получения или изменения значения которых нужны определенные права. Фундаментальная способность процесса порождать новые процессы также контролируется SELinux. Все аспекты деятельности процесса отражены в правах класса **process**.

```
$ seinfo --class process -x

Classes: 1
class process
{
        getsession
        setsockcreate
        setcurrent
        getrlimit
        execmem
        getcap
```

```
        getpgid
        signull
        execheap
        sigstop
        transition
        getattr
        setsched
        rlimitinh
        setrlimit
        dyntransition
        setpgid
        execstack
        ptrace
        setcap
        share
        siginh
        sigchld
        signal
        setexec
        noatsecure
        fork
        sigkill
        getsched
        setfscreate
        setkeycreate
}
```

5.2. Контекст безопасности

Рассмотрим контекст безопасности на примерах. В системах с интегрированным SELinux (SELinux-aware) утилиты командной строки для поддержки работы с контекстами безопасности предоставляют опцию -Z.

Выведем контекст безопасности пользователя. В политике **targeted** для выполнения пользовательских процессов предусмотрен так называемый неограниченный домен **unconfined_t**.

```
$ id -Z
unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
```

Серверные процессы выполняются в своих собственных доменах. Запустим web-сервер Apache и выведем контекст безопасности процесса **httpd**:

```
# systemctl start httpd
# ps -ZC httpd
LABEL                              PID TTY          TIME
CMD
system_u:system_r:httpd_t:s0      6799 ?       00:00:00
httpd
```

Теперь выведем контексты безопасности часто используемых файлов и директорий. Обратите внимание, что у всех объектов одна и та же роль **object_r**, так как для объектов понятие роли не имеет смысла.

```
$ ls -dZ /var/www/html/
system_u:object_r:httpd_sys_content_t:s0
/var/www/html/
$ ls -dZ /tmp/
system_u:object_r:tmp_t:s0 /tmp/
$ ls -dZ /etc/passwd /etc/shadow
system_u:object_r:passwd_file_t:s0 /etc/passwd
system_u:object_r:shadow_t:s0 /etc/shadow
$ ls -Z /home/
unconfined_u:object_r:user_home_dir_t:s0 user1
```

5.3. Контекст безопасности файла

С каждым файлом связан свой контекст безопасности, который хранится в его расширенных атрибутах. С позиции ядра Linux, SELinux является модулем LSM. Для хранения информации LSM используется пространство **security**, а для хранения контекста безопасности SELinux в пространстве **security** используется запись **selinux**.

Рассмотрим разные способы получения контекста безопасности файла на примере файла **/etc/shadow**:

```
$ ls -Z /etc/shadow
system_u:object_r:shadow_t:s0 /etc/shadow
$ stat --format='%C %n' /etc/shadow
```

```
system_u:object_r:shadow_t:s0 /etc/shadow
$ getfattr -n security.selinux /etc/shadow
getfattr: Removing leading '/' from absolute path names
# file: etc/shadow
security.selinux="system_u:object_r:shadow_t:s0"
```

Если возникла необходимость переустановить контексты безопасности для всех файлов, нужно создать файл **/.autorelabel** и перезагрузить систему.

5.4. Принудительная типизация

Базовым механизмом управления доступом в SELinux является механизм принудительной типизации (type enforcement, TE).

```
$ ls -Z /bin/ls
system_u:object_r:bin_t:s0 /bin/ls
```

По соглашению, имена типов оканчиваются на **_t**. Разработчик политики безопасности может не следовать этому соглашению, но тогда читаемость исходного кода сразу ухудшится.

Механизм TE решает две основные задачи:

- гарантия того, что доступ к файлам могут получить только определенные программы, независимо от того, какие пользователи их запускают;
- переход домена.

5.5. Предоставление доступа: правило allow

В SELinux по умолчанию любой доступ запрещен независимо от идентификаторов процесса. Здесь нет аналога пользователя **root** в DAC. Каждый доступ должен быть разрешен явно. Для этого в политике безопасности предназначено правило **allow**.

Правило **allow** состоит из четырех частей:

1. Тип-источник (source type) – обычно домен процесса, осуществляющего доступ.

2. Целевой тип (target type) – тип объекта, к которому процесс осуществляет доступ.

3. Класс объекта (object class) – класс объекта, к которому предоставляется доступ.

4. Права доступа (permissions) – виды доступа, разрешенные из домена-источника к целевому типу для данного класса объектов.

В одном правиле можно перечислить несколько доменов, целевых типов, классов объектов и прав доступа. В качестве примера рассмотрим следующее правило:

```
allow unconfined_t bin_t : file { getattr read execute };
```

В этом примере показан базовый синтаксис правила **allow**. В этом правиле используются два идентификатора типа: домен-источник (субъект доступа) **unconfined_t** и целевой тип (объект доступа) **bin_t**. Идентификатор file – это имя класса объектов, определенного в политике безопасности для представления обычного файла. Права доступа, заключенные в фигурные скобки, представляют собой подмножество видов доступа, имеющих смысл для экземпляра класса объектов **file**. Смысл этого правила следующий: процесс, выполняющийся в домене **unconfined_t**, может получать атрибуты, читать и исполнять экземпляры объекта **file** с типом **bin_t**.

Права доступа в SELinux более гранулярные, чем права DAC, где все виды доступа сводятся всего к трем (rwx). В данном примере права **read** и **execute** достаточно понятные, а вот право **getattr** менее очевидное. По сути, право **getattr** к файлу позволяет процессу получать (но не изменять) атрибуты файла, хранящиеся в его иноде (временные метки, права DAC и др.). В стандартном Linux процесс может получить метаданные файла только при наличии у него права поиска (x) на директорию, содержащую файл, даже если процесс не имеет права чтения самого файла.

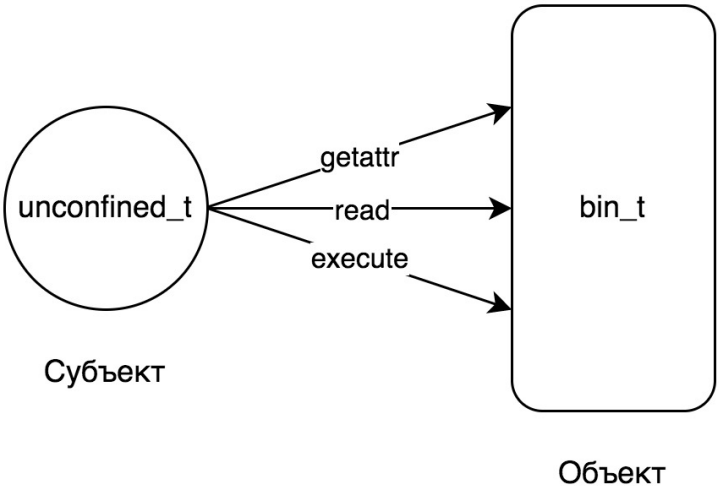
Если исходить из того, что **unconfined_t** является доменом обычного, непривилегированного пользовательского процесса, например, такого как процесс оболочки **bash**, а **bin_t** представляет тип, связанный с исполняемыми файлами, которые запускают пользователи с обычными привилегиями (например, **/bin/bash**), то тогда такое правило должно быть включено в политику безопасности, чтобы разрешить пользователям выполнять программы командной оболочки.

Для поиска правил в политике безопасности SELinux предназначена команда **sesearch**. Формат команды следующий:

```
sesearch [опции] [выражение] [файл_политики]
```

Для поиска правил **allow** нужно использовать выражение --**allow**, для указания класса объектов используется выражение **--class (-c)**, для указания домена процесса (который является источником запроса на доступ) используется выражение **--source (-s)**, для указания целевого типа – выражение **--target (-t)**.

Для графического изображения разрешенных доступов будем использовать круги – для процессов, прямоугольники – для объектов (файлов) и стрелки – для прав доступа. Например, на рис. 5.1 показан доступ, разрешенный рассмотренным правилом **allow**.




```
allow unconfined_t bin_t : file { getattr read execute };
```

Рис. 5.1. Пример правила **allow**

5.6. Пример принудительной типизации

Несмотря на простоту правила **allow**, основная сложность заключается в том, чтобы с его помощью задать тысячи доступов, необходимых для корректной работы системы в целом, при этом гарантируя, что предоставлены только необходимые права, делая таким образом работу системы настолько безопасной, насколько это возможно.

Рассмотрим работу механизма TE на примере программы управления паролями пользователей **passwd**. В GNU/Linux эта программа является привилегированной, так как в ее задачи входит модификация файла **/etc/shadow**, в котором хранятся хэши паролей пользователей.

Программа **passwd** реализует свою собственную политику безопасности, которая позволяет обычным пользователям изменять только их собственный пароль, а пользователю **root** – изменять пароль любого пользователя.

Чтобы выполнить эту важную с точки зрения информационной безопасности задачу, программа **passwd** должна иметь возможность читать и изменять файл **shadow**, а также удалять его и пересоздавать заново. Зачем нужно пересоздавать? Когда пользователь меняет пароль, измененное содержимое файла **shadow** записывается в новый файл, а старый файл удаляется. Убедиться в этом можно, посмотрев на номер инода до и после изменения пароля:

```
$ ls -i /etc/shadow
2021418 /etc/shadow
$ passwd
успешное изменение пароля
$ ls -i /etc/shadow
2022949 /etc/shadow
```

Как видно, номер инода файла **shadow** стал другим.

В стандартной системе GNU/Linux программа **passwd** может получить доступ к файлу shadow, так как владельцем исполняемого файла программы **passwd** является пользователь **root**, и для исполняемого файла установлен бит **set-user-ID**, поэтому при запуске даже обычным пользователем программа **passwd** будет исполняться

от имени пользователя **root**, который имеет полный доступ ко всем файлам в системе.

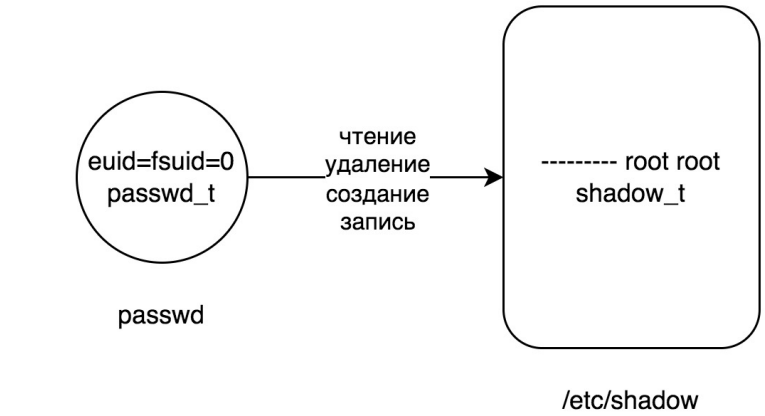
Проблема заключается в том, что пользователь **root** потенциально может запустить любую программу в системе. Из этого следует, что каждая программа имеет потенциальную возможность изменить содержимое файла **shadow**. Механизм TE позволяет гарантировать, что к файлу **shadow** сможет получить доступ только программа **passwd**, независимо от того, какой пользователь ее запустил.

На рис. 5.2 показана работа программы **passwd** под контролем механизма TE. В этом примере определено два типа. Тип **passwd_t** – это домен, предназначенный для выполнения программы **passwd**. Тип **shadow_t** – это тип для файла **shadow**. Чтобы проверить, как это выглядит на практике, запустим в одном терминале программу **passwd**, а в другом – выполним следующие команды:

```
$ ls -lZ /etc/shadow
----------. 1 root root system_u:object_r:shadow_t:s0
1419 Oct 30 04:23 /etc/shadow
$ ps -o euser,comm,label $(pgrep passwd)
EUSER    COMMAND         LABEL
root        passwd                 unconfined_u:uncon-
fined_r:passwd_t:s0-s0:c0.c1023
$ sesearch --allow -s passwd_t -t shadow_t -c file
allow passwd_t shadow_t:file { append create getattr
ioctl link lock map open read relabelfrom relabelto
rename setattr unlink write };
```

Команда **pgrep** возвращает идентификатор процесса по имени запущенной программы, а команда **sesearch** позволяет искать правила в политике безопасности по разным критериям.

Видно, что файл **shadow** принадлежит пользователю **root** и имеет тип **shadow_t**, а процесс **passwd** выполняется от имени пользователя **root** в домене **passwd_t**. В политике безопасности присутствует правило **allow**, которое разрешает доступ (на чтение, удаление, создание, запись и т.д.) из домена процесса **passwd** (**passwd_t**) к типу файла **shadow** (**shadow_t**).




```
allow passwd_t shadow_t:file { append create getattr ioctl link lock
map open read relabelfrom relabelto rename setattr unlink write };
```

Рис. 5.2. Пример работы механизма TE

Таким образом, процесс, выполняющий программу **passwd**, может управлять файлом **shadow**, так как его эффективный идентификатор равен нулю (т.е. он выполняется с правами пользователя **root**), и правило **allow** предоставляет ему необходимые виды доступа к типу файла **shadow**. Как видно, необходимо, чтобы оба механизма управления доступом – DAC и SELinux – разрешили доступ, одного из них будет недостаточно.

5.7. Проблема перехода домена

Рассмотрим вторую важную задачу, которую решает механизм TE. Кроме написания многочисленных правил доступа процессов к файлам, существует более сложная задача – гарантированный запуск только определенной программы в контексте процесса, выполняющегося только в определенном домене. Нельзя допустить, чтобы при каких-то обстоятельствах программы, которые не предназначены для работы с файлом **shadow**, могли выполняться процессом в домене **passwd_t**. Возникает проблема перехода домена (domain transition), показанная на рис. 5.3.

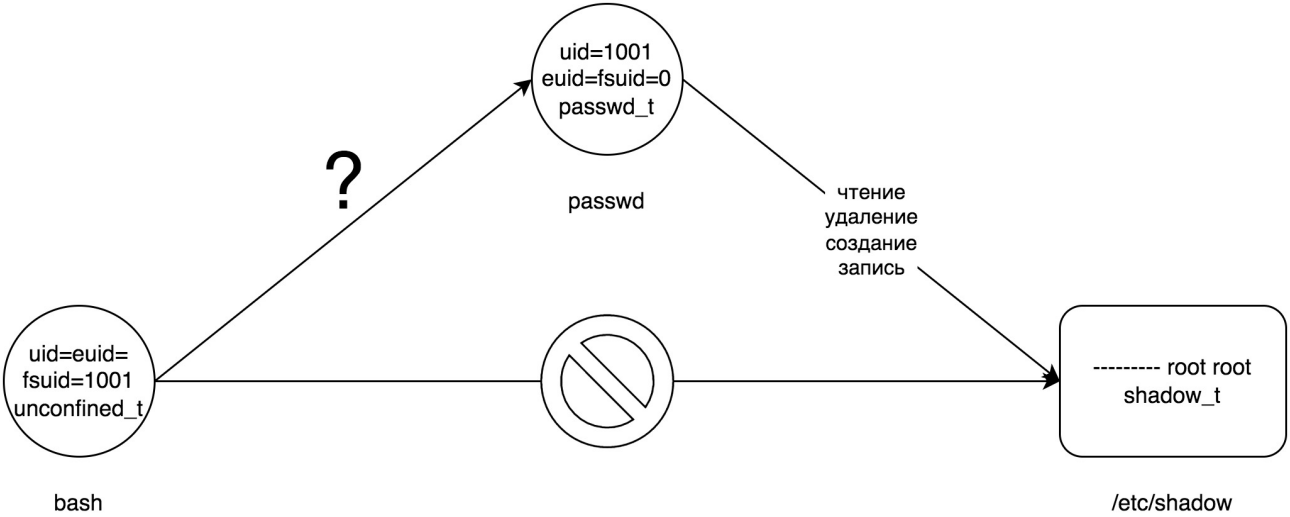


Рис. 5.3. Проблема перехода домена

После входа пользователя user1 в систему запускается командная оболочка **bash**. В стандартной системе GNU/Linux реальный и эффективный пользовательские идентификаторы одинаковы (реальный идентификатор берется из файла **/etc/passwd** по имени пользователя user1). Процесс **bash** выполняется в домене **unconfined_t**, который предназначен для обычных, непривилегированных пользовательских процессов. При запуске пользователем user1 новых программ новые процессы будут наследовать домен **unconfined_t**. Возникает вопрос, как пользователю user1 запустить программу **passwd** в домене **passwd_t**?

Нельзя допустить, чтобы процесс в домене **unconfined_t** имел возможность читать и писать в файл **shadow**, так как это позволило бы любой программе, запущенной пользователем user1, получить полный доступ к этому критически важному файлу. Только программа **passwd** должна иметь такой доступ и только тогда, когда она выполняется в домене **passwd_t**. Возникает задача поиска доверенного способа перехода из домена **unconfined_t** в домен **passwd_t**.

5.8. Переход домена

Концепция перехода домена очень похожа на концепцию повышения привилегий с помощью бита **set-user-ID** в DAC, рассмотренную в гл. 3. Рассмотри рис. 5.4.

В этом примере показаны четыре типа: домен **unconfined_t** для процесса **bash**, домен **passwd_t** для процесса **passwd**, тип **shadow_t** для файла **shadow** и новый тип **passwd_exec_t** для исполняемого файла **passwd**. Проверить на практике можно следующей командой:

```
$ ls -lZ /usr/bin/passwd
-rwsr-xr-x.     1     root     root     system_u:ob-
ject_r:passwd_exec_t:s0   37600   Jan   30      2020
/usr/bin/passwd
```

Указанные четыре типа нужны для формирования правил TE, которые обеспечивают запуск программы **passwd** в домене **passwd_exec_t**. Рассмотрим правила, необходимые для выполнения перехода домена.

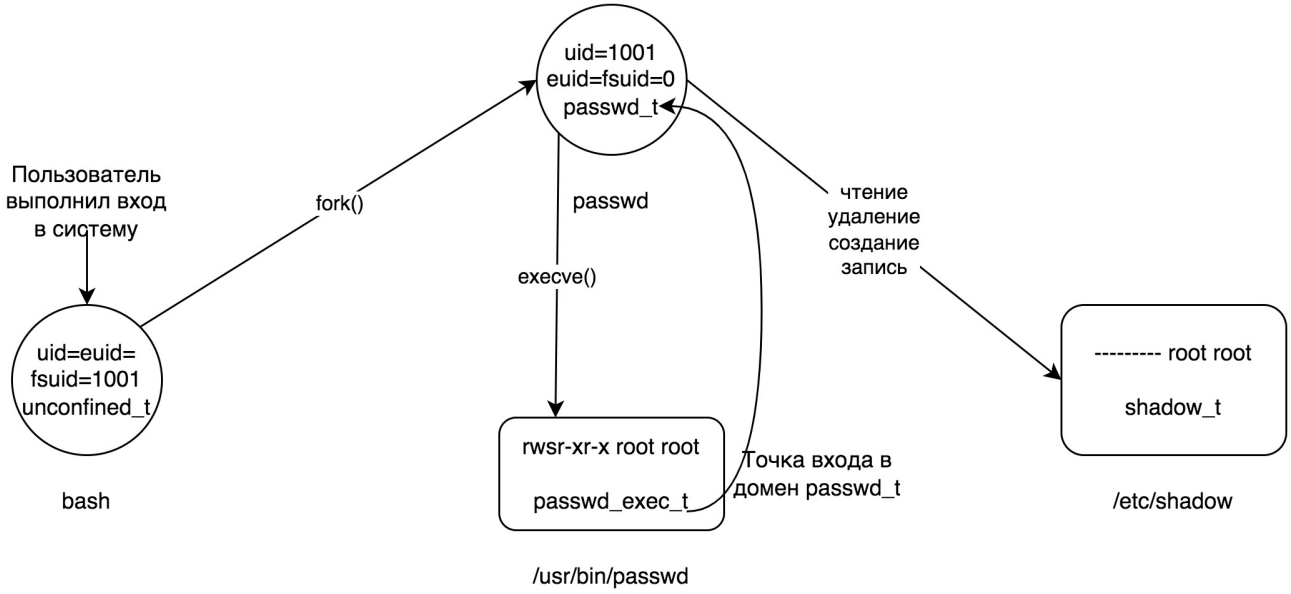


Рис. 5.4. Перехода домена

Правило 1. Позволяет процессу **bash** в текущем домене unconfined_t выполнить системный вызов **execve()** с исполняемым файлом **passwd** типа **passwd_exec_t** в качестве аргумента. Право **execute** является аналогом права на исполнение x в DAC. Так как **bash** выполняет системный вызов **stat()** к исполняемому файлу перед тем, как его запустить, нужно право **getattr**. После того, как **bash** выполнит системный вызов **fork()**, будет создан новый процесс, который унаследует от **bash** все его атрибуты безопасности, включая домен **unconfined_t**. Поэтому право execute должно относиться к текущему домену, т.е. домену процесса **bash**, и в данном правиле типом-источником является домен **unconfined_t**. Найдем это правило в политике безопасности SELinux:

```
# sesearch -A -s unconfined_t -t passwd_exec_t -c file
    allow unconfined_t passwd_exec_t:file { execute
execute_no_trans getattr ioctl map open read };
```

Правило 2. Определяет исполняемый файл с типом **passwd_exec_t** в качестве точки входа (entry point) в целевой домен **passwd_t**. Право **entrypoint** является специфическим для SELinux: оно определяет, какие исполняемые файлы (какие программы) могут «войти» в домен. Для того чтобы перейти из текущего (старого) домена в целевой (новый) домен, целевой домен должен иметь право **entrypoint** к исполняемому файлу, который используется для перехода в целевой домен. В данном примере целевым доменом является **passwd_t**. Найдем это правило в политике безопасности SELinux:

```
# sesearch -A -s passwd_t -t passwd_exec_t -c file
    allow passwd_t passwd_exec_t:file { entrypoint ex-
ecute getattr ioctl lock map open read };
```

Предположим, что только исполняемому файлу **passwd** присвоен тип **passwd_exec_t**, и что только домен **passwd**_t имеет право **entrypoint** к типу **passwd_exec_t**. Тогда получается, что только программа **passwd** может выполняться в домене **passwd_t**. Таким образом, механизм TE обеспечивает очень жесткий контроль за запуском программ!

Правило 3. До настоящего момента в правилах **allow** предоставлялся доступ к объектам класса **file**, представляющего обычные файлы. В данном правиле классом объекта является **process**, представляющий процессы. Право **transition** нужно, чтобы изменить домен в контексте безопасности процесса. Найдем это правило в политике безопасности SELinux:

```
# sesearch -A -s unconfined_t -t passwd_t -c process
    allow unconfined_t passwd_t:process transition;
```

Текущий (старый) домен **unconfined_t** должен иметь право **transition** к целевому (новому) домену, чтобы переход домена был разрешен.

Рассмотренные правила вместе обеспечивают необходимый доступ для выполнения перехода домена. Чтобы переход домена выполнился успешно, необходимы все три правила; по отдельности ни одно из правил не является достаточным. Поэтому переход домена разрешен только при выполнении следующих трех условий:

1. Текущий домен процесса имеет право на выполнение **execute** к типу файла, который является точкой входа в целевой домен.
2. Целевой домен процесса имеет право **entrypoint** к типу исполняемого файла.
3. Текущий домен процесса имеет право **transition** к целевому домену процесса.

Еще раз обратим внимание на то, что использование права **entrypoint** в отношении исполняемых файлов позволяет жестко контролировать, какие программы могут быть запущены в данном домене.

5.9. TE на примере web-сервера Apache

Рассмотрим работу механизма TE на примере web-сервера Apache. Исполняемый файл web-сервера Apache – **/usr/sbin/httpd**. Основной конфигурационный файл – **/etc/httpd/conf/httpd.conf**. Журналы хранятся в директории **/etc/httpd/logs/**. По умолчанию содержимое web-сайта размещается в директории **/var/www/html/**.

Web-сервер принимает подключение клиентов на 80 порту для протокола **http** и на 443 порту – для протокола **https**.

Запустим web-сервер Apache и выведем контекст безопасности процесса **httpd**:

```
# systemctl start httpd.service
# systemctl status httpd.service
● httpd.service - The Apache HTTP Server
Loaded:   loaded   (/usr/lib/systemd/system/httpd.ser-
vice; disabled; vendor preset: disabled)
Active: active (running) since Sat 2021-03-27 03:10:19
MSK; 10s ago
Docs: man:httpd.service(8)
Main PID: 8495 (httpd)
...
# ps -Z 8495
LABEL                                PID TTY      STAT
TIME COMMAND
system_u:system_r:httpd_t:s0        8495 ?          Ss
0:00 /usr/sbin/httpd -DFOREGROUND
```

Видно, что процесс **httpd** выполняется в домене **httpd_t**. В соответствии с концепцией TE в этом домене должны быть доступны все ресурсы, перечисленные выше. И, конечно, в политике безопасности не должно быть правил, разрешающих доступ к этим ресурсам из других доменов, отличных от **httpd_t**.

Рассмотрим контексты безопасности исполняемого файла **httpd** и ресурсов, которые требуются web-серверу Apache для своей работы:

```
# ls -Z /usr/sbin/httpd
system_u:object_r:httpd_exec_t:s0 /usr/sbin/httpd
# ls -Z /etc/httpd/conf/httpd.conf
system_u:object_r:httpd_config_t:s0
/etc/httpd/conf/httpd.conf
# ls -1Z /etc/httpd/logs/{access,error}_log
system_u:object_r:httpd_log_t:s0
```

```
/etc/httpd/logs/access_log
system_u:object_r:httpd_log_t:s0  /etc/httpd/logs/er-
ror_log
# ls -Z /var/www/html/index.html
unconfined_u:object_r:httpd_sys_content_t:s0
/var/www/html/index.html
```

В политике безопасности должны быть правила, разрешающие доступ из домена **httpd_t** к файлам с ТЕ-типами **httpd_config_t** и **httpd_sys_content_t** на чтение, а также к файлам с ТЕ-типом **httpd_log_t** на запись.

Попробуем найти соответствующие правила:

```
# sesearch --allow --source httpd_t --class file | grep
httpd_config_t
allow httpd_t httpd_config_t:file { getattr ioctl lock
map open read };
# sesearch --allow --source httpd_t --class file | grep
httpd_log_t
allow httpd_t httpd_log_t:file { create open read
setattr };
```

Чтобы связать номер порта с доменом, вводится ТЕ-тип **http_port_t**, с которым связываются стандартные номера портов, такие как 80 и 443. Поэтому в политике безопасности должно быть правило, разрешающее привязывать сервер к номеру порта.

```
# semanage port -l | grep '^http_port_t'
http_port_t tcp 80, 81, 443, 488, 8008, 8009, 8443,
9000
# sesearch --allow --source httpd_t --class tcp_socket
| grep http_port_t
allow httpd_t http_port_t:tcp_socket name_bind;
```

Данное правило разрешает процессу в домене **httpd_t** «связывать имя» (право name_bind) для класса объектов **tcp_socket** с ТЕ-типом **http_port_t**.

5.10. Конфигурирование SELinux

В конфигурационном файле **/etc/selinux/config** устанавливается режим работы SELinux и загружаемая в ядро политика безопасности. SELinux может быть отключен полностью, когда политика безопасности не загружена в ядро, или быть включенным и работать в одном из двух режимов. В системе может быть установлено несколько политик безопасности, но загружена в ядро должна быть только одна. Поэтому, если SELinux включен, необходимо передать имя политики безопасности, которая станет активной. Режим работы задается директивой SELINUX, а имя политики безопасности – директивой SELINUXTYPE. В табл. 5.1 показаны значения директивы SELINUX.

Таблица 5.1

Значения директивы SELINUX

Значение	Описание
enforcing	SELinux включен, решения о доступе форсируются, операции доступа логируются. Это боевой режим
permissive	SELinux включен, решения о доступе не форсируются, только логируются попытки доступа (как успешные, так и неуспешные). Этот режим применяется для отладки новых модулей политики безопасности
disabled	SELinux отключен, политика безопасности не загружена в ядро, операции доступа не логируются

Существует три способа управления режимом работы SELinux:

1. С помощью файла **/etc/selinux/config**. Этот способ требует перезагрузки после изменения файла.
2. С помощью команды **setenforce** или специальных файлов **/sys/fs/selinux/enforce** и **/sys/fs/selinux/disable**.
3. С помощью параметров ядра, которые передаются ядру на этапе начальной загрузки (обычно через загрузчик GRUB2).

Для получения текущего режима работы предназначена команда **getenforce**. Рассмотрим пример использования командного интерфейса:

```
# getenforce
Enforcing
# cat /sys/fs/selinux/enforce
1# setenforce Permissive
# getenforce
Permissive
# cat /sys/fs/selinux/enforce
0# echo '1' > /sys/fs/selinux/enforce
# getenforce
Enforcing
# cat /sys/fs/selinux/enforce
1# echo '0' > /sys/fs/selinux/disable SELinux
будет отключен
```

Для использования режима работы в shell-скрипте в качестве условия предназначена команда **selinuxenabled**, которая возвращает код возврата равный нулю, если SELinux находится в режиме Enforcing, и единице – если в Permissive. Пример:

```
$ getenforce
Enforcing
$ selinuxenabled
$ echo $?
0
```

Команду **selinuxenabled** удобно использовать в условных конструкциях командной оболочки.

Контрольные вопросы

1. Для чего предназначены классы объектов?
2. На какие группы можно разделить права доступа SELinux?
3. Что общего между классами объектов обычного файла **file** и директории **dir**?
4. Что включает контекст безопасности?

5. Как используется роль SELinux для объектов?
6. Где хранится контекст безопасности файла?
7. Каким способом можно исследовать правила политики безопасности?
8. Какие основные задачи решает механизм принудительной типизации?
9. При выполнении каких условий разрешен переход домена?
10. В каких режимах может функционировать SELinux?

Глава 6. Идентификация и аутентификация

В данной главе рассматриваются механизмы идентификации и аутентификации, которые формально не относятся к механизмам ядра операционной системы, а существуют в пространстве пользователя, но при этом представляют важное значение с точки зрения безопасности операционной системы.

6.1. Концепция пользователя

Как уже было сказано, Linux является многопользовательской операционной системой, в которой одновременно могут работать несколько пользователей. Для реализации этой концепции вводится понятие *учетной записи пользователя* (user account), или просто *пользователя*. С точки зрения DAC каждый файл принадлежит определенному пользователю, а каждый процесс выполняется от имени определенного пользователя. Таким образом, концепция пользователя позволяет определить владельцев системных ресурсов, права доступа к этим ресурсам и связать внутренние сущности системы с внешним миром.

Для примера выведем идентификаторы пользователя, затем создадим новый файл и выведем владельцев нового файла и файла **/etc/passwd**, затем посмотрим, от имени какого пользователя выполняется командная оболочка **bash** (поле UID):

```
$ id
uid=1001(user1) gid=1001(user1) groups=1001(user1)
context=unconfined_u:unconfined_r:unconfined_t:s0-
s0:c0.c1023
$ touch file1
$ ls -l file1
-rw-rw-r--. 1 user1 user1 0 Oct  2 09:44 file1
$ ls -l /etc/passwd
-rw-r--r--. 1 root root 2740 Apr 22 00:42 /etc/passwd
$ ps -l
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN
TTY          TIME CMD
```

```
4 S  1001    6995    6994  0  80   0 - 56851 -
pts/0    00:00:00 bash
0 R  1001    7035    6995  0  80   0 - 54699 -
pts/0    00:00:00 ps
```

В Linux конфигурационные файлы, как правило, представляют собой обычные текстовые файлы, что позволяет производить настройку системы с помощью штатных средств командной строки. Для управления пользователями используются следующие файлы: **/etc/passwd**, **/etc/shadow**, **/etc/group**, **/etc/default/useradd**, **/etc/login.defs**. Рассмотрим эти файлы.

Список пользователей системы хранится в текстовом файле **/etc/passwd**. Каждая строка этого файла описывает одного пользователя и представляет собой запись, состоящую из семи полей, разделенных двоеточием. В табл. 6.1 представлены порядок и назначение полей, а также опции команды **useradd**, которые позволяют управлять соответствующими полями.

Таблица 6.1

Формат записей файла **/etc/passwd**

Номер поля	Назначение	Опции команды **useradd**
1	Логическое (входное) имя пользователя. Имя может состоять из маленьких букв, цифр, символов подчеркивания или тире. Имя должно начинаться с маленькой буквы или символа подчеркивания и может заканчиваться символом доллара. Регулярное выражение для имени пользователя следующее: [a-z_][a-z0-9_-]*[$]? Имя может быть длиной до 32 символов	Последний аргумент команды
2	Не используется, исторически в UNIX в этом поле хранился хэш пароля пользователя	–
3	UID (user id) – идентификатор пользователя	-u

Окончаните табл. 6.1

Номер поля	Назначение	Опции команды **useradd**
4	GID (group id) – идентификатор основной группы пользователя	-g
5	Комментарий, например, настоящее имя пользователя. Иногда это поле называют GECOS (от General Electric Comprehensive Operating System), так как в первых версиях UNIX в этом поле хранился идентификатор для доступа к серверу печати, который работал под управлением GECOS	-c
6	Путь к домашней директории пользователя, в которую попадает пользователь после входа в систему и владельцем которой он является	-d
7	Путь к программе, которая будет запускаться после входа пользователя в систему. Обычно это путь к интерпретатору команд **bash**	-s

Принято, что домашняя директория пользователя **root** – это **/root**, а для обычных пользователей домашние директории располагаются в директории **/home**.

Во время входа пользователя в систему процесс входа **login** становится процессом **bash**, в котором в качестве рабочей директории устанавливается домашняя директория пользователя.

В качестве примера работы с файлом **/etc/passwd** рассмотрим shell-скрипт, который выводит на экран информацию о пользователях, у которых в качестве оболочки установлен **bash** [21]. Для разбиения записей на отдельные поля используется тот факт, что поля разделяются символом ‘:’. Поэтому если переопределить переменную IFS, то BASH будет читать отдельные поля, как отдельные слова, разделенные символом-разделителем.

Листинг 6.1. Вывод информации о пользователях.

```
grep 'bash$' /etc/passwd |
while IFS=: read user passwd uid gid name homedir shell
do
    printf "%16s: %s\n" \
        User             "$user" \
        Password         "$passwd" \
        "User ID"        "$uid" \
        "Group ID"       "$gid" \
        Name             "$name" \
        "Home directory" "$homedir" \
        Shell            "$shell"
    read < /dev/tty
done
```

Для управления паролями пользователей используется текстовый файл **/etc/shadow**, который принадлежит пользователю **root** и не доступен на чтение и запись обычным пользователям. Каждая строка этого файла соответствует одному пользователю системы и представляет собой запись, состоящую из девяти полей, разделенных двоеточием. В табл. 6.2 представлены порядок и назначение полей файла **/etc/shadow**.

Основная функция файла **/etc/shadow** – хранить хэши паролей пользователей системы. Для вычисления хэша пароля должна использоваться криптографическая хэш-функция. Определение и свойства криптографической хэш-функции приведены в отечественном стандарте ГОСТ Р 34.11-2012 [3]:

Хэш-функция (hash-function) – это функция, отображающая строки бит в строки бит фиксированной длины. Криптографическая хэш-функция должна удовлетворять следующим свойствам:

1. Стойкость к вычислению прообраза, т.е. по данному значению функции сложно вычислить исходные данные, отображаемые в это значение;
2. Стойкость к вычислению второго прообраза, т.е. для заданных исходных данных сложно вычислить другие исходные данные, отображаемые в то же значение функции;
3. Стойкость к поиску коллизий, т.е. сложно вычислить какую-либо пару исходных данных, отображаемых в одно и то же значение.

Рассмотрим пример содержимого второго поля:

```
:$6$ZjdXqIs4BlbjqkBs$7oRI476rdxWEriysR0UYScdvaLrHO2uq
9msYpVxnxeO0zuaeDTvpmfedb8oLC8kDBz9FPrrzRLn70y9//f06x
.:
```

В поле, содержащем хэш пароля, можно выделить три части, разделенные знаком $: алгоритм хэширования (цифра 6 означает SHA-512); так называемую *соль* – случайное число, которое добавляется к паролю, чтобы вычислить хэш (таким образом, даже если у двух пользователей будут одинаковые пароли, их хэши будут разными); сам хэш.

Таблица 6.2

Формат записей файла **/etc/shadow**

Номер поля	Назначение
1	Логическое (входное) имя существующего пользователя
2	Хэш пароля. Если поле пустое, то аутентификация для пользователя не требуется. Если содержимое поля начинается с восклицательного знака, то считается, что пользователь заблокирован, т.е. он не сможет войти в систему с помощью пароля. Однако пользователь может войти с помощью других средств (например, ssh)
3	Дата последнего изменения пароля, в количестве дней, прошедших с 1 января 1970 г. Значение «0» показывает, что пользователь должен изменить свой пароль при следующем входе в систему. Пустое поле означает, что пароль никогда не устаревает
4	Минимальный срок действия пароля – количество дней, которые должны пройти, чтобы пользователь смог снова изменить свой пароль. Пустое поле или значение «0» показывают, что минимальный срок не установлен

Окончание табл. 6.2

Номер поля	Назначение
5	Максимальный срок действия пароля – количество дней, по прошествии которых пользователь должен изменить свой пароль. По прошествии этого количества дней у пользователя будет запрошено изменение пароля при следующем входе в систему. Пустое поле означает, что не установлены максимальный срок, период предупреждения, период бездействия. Если максимальный срок действия пароля меньше минимального срока действия пароля, пользователь не сможет изменить свой пароль
6	Период предупреждения – количество дней до истечения максимального срока действия пароля, в течение которого пользователю выводятся предупреждения о необходимости поменять пароль. Пустое поле или значение «0» означают, что период предупреждения не установлен
7	Период бездействия пароля – количество дней после истечения максимального срока действия пароля, в течение которого учетная запись продолжает быть активной, и пользователь может войти в систему, чтобы изменить свой пароль. После истечения периода бездействия вход пользователя в систему заблокирован, учетная запись становится неактивной. Пользователь должен обратиться к администратору. Пустое поле означает, что период бездействия не установлен
8	Срок действия учетной записи в количестве дней, прошедших с 1 января 1970 г. Пустое поле означает, что срок действия учетной записи никогда не истечет. Значение «0» использовать не следует, так как оно может интерпретироваться как учетная запись без срока действия или с истечением срока действия 1 января 1970 г.
9	Зарезервировано для будущего использования

Соотношение временных характеристик пароля показано на рис. 6.1.

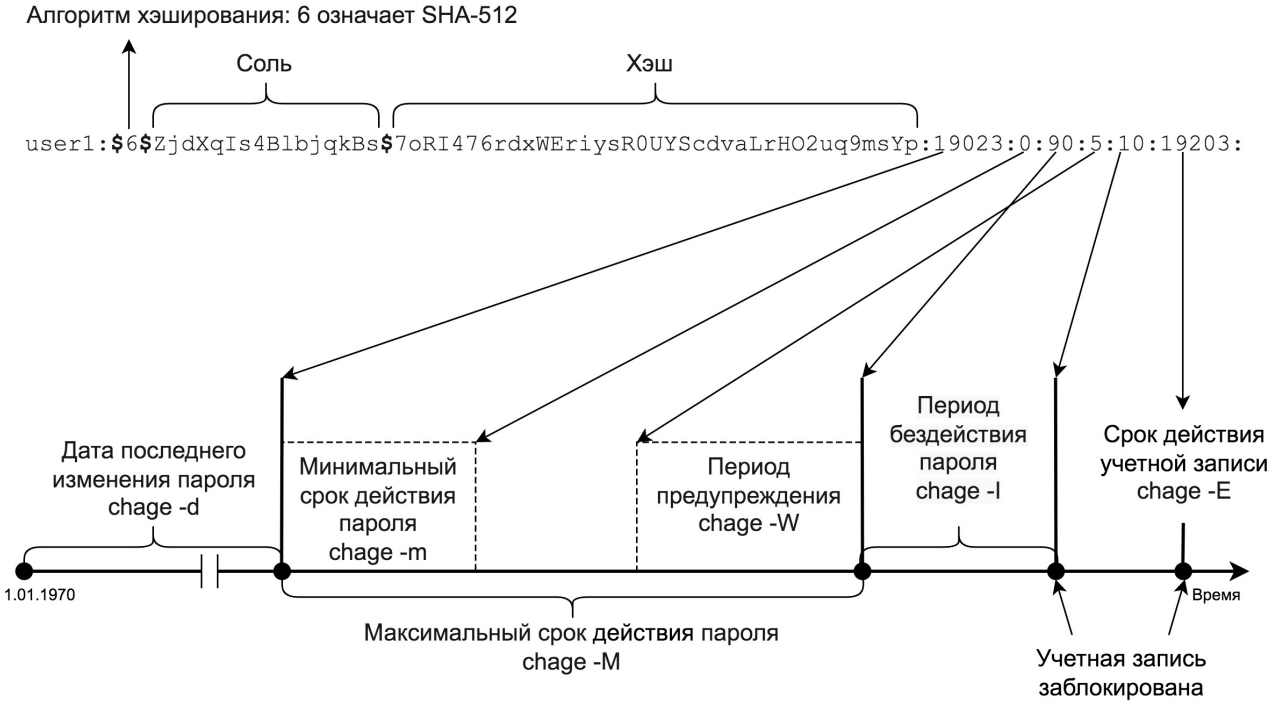


Рис. 6.1. Характеристики пароля пользователя

Обратите внимание, что срок действия учетной записи и срок действия пароля – две разных настройки. В случае истечения срока действия учетной записи пользователю полностью запрещен вход в систему. В случае истечения срока действия пароля пользователю не разрешено входить в систему, используя свой пароль.

Рассмотрим пример. Предположим, что сегодня 31 января, и в системе создан пользователь, который будет работать над проектом в течение шести месяцев (180 дней). Команда **date** позволяет вычислить дату, которая наступит через определенное количество дней после текущего дня. Срок действия учетной записи для нового пользователя можно ограничить 180 днями. Пароль требуется менять не реже, чем каждые три месяца (90 дней). Минимальный срок действия пароля не установлен. За пять дней до обязательной смены пароля пользователю начнут выдаваться предупреждения. Если пользователь не поменяет свой пароль, то в течение 10 дней он сможет войти в систему со старым паролем и установить новый пароль. Команда **chage -l** выводит текущие настройки пароля пользователя.

```
# date
Mon 31 Jan 2022 02:50:12 PM MSK
# date -d +180days +%Y-%m-%d
2022-07-30
# chage -E $(date -d +180days +%Y-%m-%d) user1
# chage -m 0 -M 90 -W 5 -I 10 user1
# chage -l user1
Last password change    : Jan 31, 2022
Password expires        : May 01, 2022
Password inactive : May 11, 2022
Account expires         : Jul 30, 2022
Minimum number of days between password change  : 0
Maximum number of days between password change  : 90
Number of days of warning before password expires
      : 5
# grep user1 /etc/shadow
user1:$6$ZjdXqIs4BlbjqkBs$7oRI476rdxWEr-
iysR0UYScdvaLrHO2uq9msYpVxnxeO0zuaeDTvpmfedb8oLC8kDBz
9FPrrzRLn70y9//f06x.:19023:0:90:5:10:19203:
```

Видно, что в записи для пользователя в файле **/etc/shadow** установлены соответствующие поля. Например, число 19023 означает количество дней с 1 января 1970 г. по 31 января 2022 г.

Если нужно заставить пользователя сменить свой пароль при следующем входе в систему, то можно воспользоваться командой **chage -d 0**.

```
# chage -d 0 user1
# chage -l user1
Last password change    : password must be changed
Password expires        : password must be changed
Password inactive : password must be changed
Account expires         : Jul 30, 2022
Minimum number of days between password change  : 0
Maximum number of days between password change  : 90
Number of days of warning before password expires
      : 5
# grep user1 /etc/shadow
user1:$6$ZjdXqIs4BlbjqkBs$7oRI476rdxWEr-
iysR0UYScdvaLrHO2uq9msYpVxnxeO0zuaeDTvpmfedb8oLC8kDBz
9FPrrzRLn70y9//f06x.:0:0:90:5:10:19203:
```

Видно, что в записи для пользователя в файле **/etc/shadow** в поле даты последнего изменения пароля установлен 0.

Пользователи объединяются в группы для более гибкого управления доступом к файлам. Пользователь должен входить как минимум в одну группу – такую группу называют ***основной*** (***primary***). Также пользователь может входить в несколько дополнительных (supplementary) групп. Список групп хранится в текстовом файле **/etc/group**. Каждая строка этого файла описывает одну группу и представляет собой запись, состоящую из четырех полей, разделенных двоеточием. В табл. 6.3 представлены порядок и назначение полей, а также опции команды **useradd**, которые позволяют управлять соответствующими полями.

Таблица 6.3

Формат записей файла **/etc/group**

Номер поля	Назначение	Опции команды **useradd**
1	Имя группы. Имя может состоять из маленьких букв, цифр, символов подчеркивания или тире. Имя должно начинаться с маленькой буквы или символа подчеркивания и может заканчиваться символом доллара. Регулярное выражение для имени группы следующее: [a-z_][a-z0-9_-]*[$]? Имя группы может быть длиной до 16 символов	Последний аргумент команды
2	Хэш пароля. Встречается редко	–
3	GID (group id) – идентификатор группы	-g
4	Список имен пользователей, входящих в группу, через запятую	-U

6.2. Управление учетной записью пользователя

Для создания, модификации и удаления учетных записей пользователей система предоставляет как командный интерфейс, так и набор библиотечных функций. Но так как информация о пользователе хранится в простых текстовых файлах, управлять учетными записями можно и вручную. Рассмотрим последовательность шагов для создания учетной записи пользователя вручную:

1. Выбрать отличительные характеристики пользователя: логическое имя, идентификатор пользователя UID, идентификатор группы пользователя GID.
2. Создать запись в **/etc/passwd**.
3. Создать запись в **/etc/group**.
4. Создать домашнюю директорию пользователя по шаблону: /home/<имя_пользователя>.

5. Скопировать в домашнюю директорию файлы, которые должны присутствовать у пользователя и, в том числе, выполняться при входе пользователя в систему.

6. Изменить владельца и группу домашней директории со всем ее содержимым на вновь созданного пользователя и его группу. При необходимости изменить права к домашней директории.

7. Создать запись в **/etc/shadow** в соответствии с политикой безопасности организации по управлению паролями (например, как часто надо пользователю менять свой пароль).

8. Установить пользователю пароль.

6.3. Командный интерфейс

Командный интерфейс управления учетными записями пользователей представлен следующими командами:

1. Команды управления пользователями: **useradd**, **userdel**, **usermod**.

2. Команды управления группами: **groupadd**, **groupdel**, **groupmod**.

3. Команды управления паролями: **passwd**, **chage**.

Рассмотрим только основные команды. Для создания пользователя предназначена команда **useradd**. Синтаксис команды:

```
useradd [опции] имя_пользователя
```

С помощью опций можно явно задать каждую характеристику пользователя (см. табл. 6.1). Однако, если опции не заданы, берутся значения по умолчанию из файлов **/etc/default/useradd** и **/etc/login.defs**. Кроме того, после создания домашней директории пользователя в нее копируется содержимое директории **/etc/skel/**, в которую системный администратор может поместить файлы, которые по умолчанию должны присутствовать у каждого пользователя.

Рассмотрим содержимое файла **/etc/default/useradd**:

```
$ cat /etc/default/useradd
GROUP=100
HOME=/home
INACTIVE=-1
```

```
EXPIRE=
SHELL=/bin/bash
SKEL=/etc/skel
CREATE_MAIL_SPOOL=yes
```

Эту же информацию можно получить с помощью опции **-D**:

```
$ useradd -D
GROUP=100
HOME=/home
INACTIVE=-1
EXPIRE=
SHELL=/bin/bash
SKEL=/etc/skel
CREATE_MAIL_SPOOL=yes
```

Приведем пример содержимого директории **/etc/skel/**:

```
$ ls -al /etc/skel/
total 32
drwxr-xr-x.   3 root root  4096 Apr 23  2020 .
drwxr-xr-x. 150 root root 12288 Oct  9 00:42 ..
-rw-r--r--.   1 root root   18 Jun  2  2020 .bash_logout
-rw-r--r--.   1 root root   141 Jun  2  2020 .bash_pro-
file
-rw-r--r--.   1 root root   376 Jun  2  2020 .bashrc
```

Рассмотрим фрагмент файла **/etc/login.defs**:

```
$ cat /etc/login.defs
UMASK 022
HOME_MODE 0700
PASS_MAX_DAYS 99999
PASS_MIN_DAYS 0
PASS_WARN_AGE 7
UID_MIN                  1000
UID_MAX                 60000
SYS_UID_MIN               201
SYS_UID_MAX               999
ENCRYPT_METHOD SHA512
CREATE_HOME yes
```

Все пространство идентификаторов пользователя (UID) можно разделить на следующие диапазоны:

0 – идентификатор пользователя **root**, который в Linux считается полноправным администратором системы, также называемого *суперпользователем*. Далее будет показано, как SELinux ограничивает права пользователя **root**.

1–999 – идентификаторы пользователей, от имени которых выполняются системные процессы.

1000–60000 – идентификаторы обычных пользователей.

Рассмотрим пример создания пользователя:

```
# useradd user1
$ grep user1 /etc/passwd
user1:x:1001:1001::/home/user1:/bin/bash
# grep user1 /etc/shadow
user1:!!:18738:0:99999:7:::
$ grep user1 /etc/group
user1:x:1001:
```

Видно, что команда **useradd** по умолчанию создает записи для нового пользователя в трех основных файлах. Для пользователя создана новая индивидуальная группа, в которую входит только данный пользователь. Пароль пользователю не установлен, поэтому его нужно установить явно командой **passwd**.

Для модификации пользователя используется команда **usermod**. Ее опции в основном повторяют опции команды **useradd**. Синтаксис команды:

```
usermod [опции] имя_пользователя
```

Для добавления пользователя в дополнительные группы используется опция **-G.** Однако, если пользователя нужно добавить в еще одну дополнительную группу, но не надо удалять из других дополнительных групп, то нужно использовать опцию **-a**. Рассмотрим пример:

```
$ grep user1 /etc/group
user1:x:1001:
```

```
# usermod -G users user1
$ grep user1 /etc/group
users:x:100:user1
user1:x:1001:
# usermod -aG wheel user1
$ grep user1 /etc/group
wheel:x:10:user1
users:x:100:user1
user1:x:1001:
```

Иногда пользователю требуется возможность аутентификации по паролю, но не требуется работать в командной оболочке. Например, на почтовом сервере пользователи должны иметь учетные записи, чтобы подключаться из почтовых клиентов. Тогда в качестве входной оболочки можно установить программу **/sbin/nologin**:

```
# usermod -s /sbin/nologin user1
# grep user1 /etc/passwd
user1:x:1001:1001::/home/user1:/sbin/nologin
```

Если пользователь попытается войти в систему, программа **/sbin/nologin** просто закроет сессию.

Для удаления пользователя используется команда **userdel**. Синтаксис команды:

```
userdel [опции] имя_пользователя
```

При удалении пользователя возникает ряд вопросов, например: что делать с его файлами как в домашней директории, так и в других частях файловой системы. Нужно ли отдавать освободившийся UID новым пользователям или нет? И т.д. По умолчанию команда **userdel** только удаляет учетную запись пользователя, но не затрагивает его файлы. Для удаления домашней директории используется опция **-r**.

Рассмотрим пример, когда удаляется только учетная запись пользователя, но после создания нового пользователя последний получает доступ к файлам пользователя, который был удален.

```
# useradd user1
$ id user1
uid=1001(user1) gid=1001(user1) groups=1001(user1)
$ ls -l /home/
drwx------.  3 user1   user1    4096 Oct  9 04:47 user1
# userdel user1
$ ls -l /home/
drwx------.  3    1001    1001  4096 Oct  9 04:47 user1
# useradd user2
$ tail -1 /etc/passwd
user2:x:1001:1001::/home/user2:/bin/bash
$ id user2
uid=1001(user2) gid=1001(user2) groups=1001(user2)
$ ls -l /home/
drwx------.  3 user2   user2    4096 Oct  9 04:47 user1
drwx------.  3 user2   user2    4096 Oct  9 04:48 user2
```

Чтобы предотвратить подобные ситуации, рекомендуется не удалять пользователя, а блокировать его учетную запись. Дополнительно можно придерживаться правил, следующих ниже.

При создании нового пользователя явно указывать его UID. Например:

```
# useradd -u 1002 user2
```

При удалении пользователя перемещать все его файлы в специально выделенную директорию, а его домашнюю директорию удалять. Например:

```
# find / -user user1 -exec mv {} /new/location;
2>/dev/null
# userdel -r user1
```

Чтобы найти файлы, которые не принадлежат никакому пользователю, можно выполнить следующую команду:

```
# find / -nouser -o -nogroup 2>/dev/null
```

Для изменения временных атрибутов пароля пользователя используется команда **chage**. Синтаксис команды:

```
chage [опции] имя_пользователя
```

Опция **-l** выводит на экран информацию об учетной записи пользователя. Опция **-d** устанавливает дату последнего изменения пароля.

Для изменения пароля пользователя, а также временных атрибутов пароля используется команда **passwd**. Синтаксис команды:

```
passwd [опции] имя_пользователя
```

Опция **-d** полностью удаляет пароль пользователя.

Чтобы заставить пользователя сменить пароль при следующем входе в систему, можно выполнить команду **passwd -e** или **chage -d 0**. В результате, в поле даты последнего изменения пароля будет записан 0.

Чтобы заблокировать вход пользователя в систему по паролю, можно использовать команды **usermod** с опцией **-L** или **passwd** с опцией **-l**.

После создания учетной записи пользователя в поле, где хранится хэш пароля, записан восклицательный знак, что не позволяет пользователю войти в систему без пароля. При блокировке пользователя хэш пароля не изменяется, но в начало записывается восклицательный знак. При разблокировании пользователя восклицательный знак просто удаляется. Рассмотрим пример:

```
# useradd user1
# grep user1 /etc/shadow
user1:!!:18909:0:90:7:::
# passwd user1
ввод пароля
# grep user1 /etc/shadow
user1:$6$4abg1ZYr.wQtQRS9$pULX4PxLl9vEQuY-
KKC/2ycmaaUVkJHFv28eKcLzrpatYZh0j0X2Urlq1/Ge-
DOIMqsK95VsKybMelYzdrmLLQn0:18909:0:90:7:::
# usermod -L user1
# grep user1 /etc/shadow
user1:!$6$4abg1ZYr.wQtQRS9$pULX4PxLl9vEQuY-
KKC/2ycmaaUVkJHFv28eKcLzrpatYZh0j0X2Urlq1/Ge-
DOIMqsK95VsKybMelYzdrmLLQn0:18909:0:90:7:::
# usermod -U user1
```

```
# grep user1 /etc/shadow
user1:$6$4abg1ZYr.wQtQRS9$pULX4PxLl9vEQuY-
KKC/2ycmaaUVkJHFv28eKcLzrpatYZh0j0X2Urlq1/Ge-
DOIMqsK95VsKybMelYzdrmLLQn0:18909:0:90:7:::
# passwd -l user1
# grep user1 /etc/shadow
user1:!!$6$4abg1ZYr.wQtQRS9$pULX4PxLl9vEQuY-
KKC/2ycmaaUVkJHFv28eKcLzrpatYZh0j0X2Urlq1/Ge-
DOIMqsK95VsKybMelYzdrmLLQn0:18909:0:90:7:::
# passwd -u user1
# grep user1 /etc/shadow
user1:$6$4abg1ZYr.wQtQRS9$pULX4PxLl9vEQuY-
KKC/2ycmaaUVkJHFv28eKcLzrpatYZh0j0X2Urlq1/Ge-
DOIMqsK95VsKybMelYzdrmLLQn0:18909:0:90:7:::
```

Как видно, возможности команд **usermod**, **passwd** и **chage** пересекаются, что позволяет системному администратору выбрать свой стиль общения с системой.

6.4. Библиотечные функции

Рассмотрим четыре функции для извлечения информации из файла **/etc/passwd**:

```
#include <sys/types.h>
#include <pwd.h>

struct passwd *getpwnam(const char *name);
struct passwd *getpwuid(uid_t uid);

int getpwnam_r(const char *restrict name, struct
passwd *restrict pwd,
                      char *restrict buf, size_t bu-
flen,
                      struct passwd **restrict re-
sult);
int getpwuid_r(uid_t uid, struct passwd *restrict
pwd,
                      char *restrict buf, size_t bu-
flen,
```

```
                        struct passwd **restrict
result);
```

Функция **getpwnam()** возвращает указатель на структуру, содержащую разбитую по полям запись для пользователя, заданного его логическим именем. Функция **getpwuid()** возвращает указатель на структуру, содержащую разбитую по полям запись для пользователя, заданного его числовым идентификатором (UID).

Структура **passwd** определена в заголовочном файле **<pwd.h>** следующим образом:

```
struct passwd {
    char   *pw_name;       /* логическое имя пользова-
теля */
    char   *pw_passwd;     /* пароль пользователя */
    uid_t   pw_uid;        /* UID */
    gid_t   pw_gid;        /* GID */
    char   *pw_gecos;     /* информация о пользователе
*/
    char   *pw_dir;        /* домашняя директория */
    char   *pw_shell;      /* командная оболочка */
    };
```

Функции **getpwnam_r()** и **getpwuid_r()** получают такую же информацию, как и функции **getpwnam()** и **getpwuid()**, но сохраняют извлеченную структуру **passwd** в буфере, на который указывает аргумент **pwd**.

Рассмотрим пример программы, приведенной в справочной странице **getpwnam(3)**. Программа демонстрирует использование библиотечной функции **getpwnam_r()** для нахождения полного имени и идентификатора пользователя, логическое имя которого передается программе в качестве первого аргумента.

Листинг 6.2. Программа p_getpwnam_r.c.

```
#include <pwd.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

```
#include <errno.h>

int
main(int argc, char *argv[])
{
    struct passwd pwd;
    struct passwd *result;
    char *buf;
    size_t bufsize;
    int s;

    if (argc != 2) {
        fprintf(stderr,   "Usage:   %s   username\n",
argv[0]);
        exit(EXIT_FAILURE);
    }

    bufsize = sysconf(_SC_GETPW_R_SIZE_MAX);
    if (bufsize == -1)          /* Value was indeter-
minate */
        bufsize = 16384;        /* Should be more than
enough */

    buf = malloc(bufsize);
    if (buf == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }

    s = getpwnam_r(argv[1], &pwd, buf, bufsize, &re-
sult);

    if (result == NULL) {
        if (s == 0)
            printf("Not found\n");
        else {
            errno = s;
            perror("getpwnam_r");
        }
        exit(EXIT_FAILURE);
    }
```

```
    printf("Name: %s; UID: %jd\n", pwd.pw_gecos,
            (intmax_t) pwd.pw_uid);
    exit(EXIT_SUCCESS);
}
```

Сначала найдем запись для пользователя user1 в файле **/etc/passwd** с помощью команды **grep**, а затем скомпилируем программу и выведем с ее помощью информацию для пользователя user1:

```
$ grep user1 /etc/passwd
user1:x:1001:1001:Студент Иванов
И.И.:/home/user1:/bin/bash
$ gcc p_getpwnam_r.c -o p_getpwnam_r
$ ./p_getpwnam_r user1
Name: Студент Иванов И.И.; UID: 1001
```

Видно, что программа вывела значение поля с описанием пользователя и его числовой идентификатор для пользователя user1.

Контрольные вопросы

1. Что входит в понятие учетной записи пользователя?
2. Какая информация хранится в файле **/etc/shadow**?
3. Чем отличается срок действия учетной записи от срока действия пароля?
4. Какими свойствами должна обладать криптографическая хэш-функция?
5. Для чего используется хэш-функция в управлении пользователями?
6. Что происходит (меняется в системе) при создании нового пользователя?
7. Как при создании пользователя формируется набор файлов, помещаемых в его домашнюю директорию?
8. Как добавить пользователя в дополнительную группу?
9. В чем особенность учетных записей на почтовом сервере?
10. Что происходит с файлами пользователя после удаления его учетной записи? Какие возникают риски безопасности?

Список литературы

1. Ефанов Д.В., Мельников В.В., Никитин В.Д. Алгоритмы и структуры ядра Linux. Учебное пособие. – М.: МИФИ, 2002.
2. ГОСТ 15971-90. Системы обработки информации. Термины и определения. – М.: Изд-во стандартов, 1991.
3. ГОСТ Р 34.11-2012. Информационная технология. Криптографическая защита информации. Функция хэширования. – М.: Стандартинформ, 2013.
4. Ryckman G.F. 17. The IBM 701 Computer at the General Motors Research Laboratories, in Annals of the History of Computing, vol. 5, no. 2, pp. 210–212, April-June 1983, doi: 10.1109/МАНС. 1983. 10026.
5. Зегжда Д.П., Ивашко А.М. Основы безопасности информационных систем. – М.: Горячая линия – Телеком, 2000.
6. Галатенко В.А. Стандарты информационной безопасности. – 2-е изд. – М.: Интернет-университет информационных технологий, 2006.
7. Грушо А.А., Тимонина Е.Е. Теоретические основы защиты информации. – М.: Изд-во «Яхтсмен», 1996.
8. Trusted Computer System Evaluation Criteria, 1983.
9. Керриск М. Linux API. Исчерпывающее руководство. – СПб.: Питер, 2019.
10. Стивенс Р., Раго С. UNIX. Профессиональное программирование. – 2-е изд. – СПб.: Символ Плюс, 2007.
11. Бовет Д., Чезати М. Ядро Linux. – 3-е изд. – СПб.: БХВ-Петербург, 2007.
12. Столлингс В. Операционные системы: внутренняя структура и принципы проектирования. – 9-е изд. – СПб.: ООО «Диалектика», 2020.
13. Silberschatz A., Galvin P.B., Gagne G. Operating system concepts, 10th edition. – NJ: Wiley, 2018.
14. Иртегов Д.В. Введение в операционные системы. – 2-е изд. – СПб.: БХВ-Петербург, 2008.
15. Аблязов Р.З. Программирование на ассемблере на платформе x86-64. – М.: ДМК Пресс, 2011.
16. The Intel 64 and IA-32 Architectures Software Developer's Manual. Vol. 3 (3A, 3B, 3C & 3D): System Programming Guide, 2016.

17. Mayer F., MacMillan K., Caplan D. SELinux by example: understanding security enhanced Linux. Upper Saddle River, NJ: Prentice Hall, 2007.

18. McCarty Bill. SELinux. NSA's Open Source Security Enhanced Linux. O'Reilly Media, 2004.

19. Vermeulen Sven. SELinux System Administration. Packt Publishing, 2013.

20. Операционные системы. Основы UNIX: Учебное пособие / А.Б. Вавренюк, О.К. Курышева, С.В. Кутепов, В.В. Макаров. – М.: ИНФРА-М, 2018.

21. Pro Bash Programming: Scripting the GNU/Linux Shell. Chris Johnson. Apress, 2009.

Приложение 1

В табл. П.1 представлены системные вызовы, рассматриваемые в данном пособии. Полный список номеров системных вызовов для архитектуры x86-64 содержится в файле **/usr/include/asm/unistd_64.h**.

Таблица П.1

Системные вызовы для архитектуры x86-64

Номер	Функция языка С	Назначение
0	read	Чтение из файла
1	write	Запись в файл
2	open	Открытие (создание) файла
3	close	Закрытие файла
4	stat	Получение метаданных файла
9	mmap	Отображение файла в память процесса
39	getpid	Получение идентификатора процесса
57	fork	Создание процесса
59	execve	Замещение программы процесса
60	exit	Завершение процесса
61	wait4	Ожидание завершения процесса-потомка
90	chmod	Изменение прав доступа к файлу
92	chown	Изменение владельца и группы – владельца файла
95	umask	Получение и изменение маски создания файлов процесса

В приложении рассматриваются особенности создания файла в директории при разных условиях.

Для эксперимента создадим новую файловую систему, но не на блочном устройстве (разделе диска), а в обычном файле. В результате в файле будет храниться образ файловой системы. Чтобы начать работать с файловой системой, нужно связать файл-образ со специальным loop-устройством, а затем ее примонтировать как обычную файловую систему на блочном устройстве.

Для создания файла-образа выполним команду **dd**, затем – команду **du**, чтобы проверить размер созданного файла:

```
# dd if=/dev/zero of=loop-fs.img bs=1M count=10
10+0 records in
10+0 records out
10485760 bytes (10 MB, 10 MiB) copied, 0.0101878 s,
1.0 GB/s
# du -h loop-fs.img
10M   loop-fs.img
```

Свяжем созданный файл с первым свободным loop-устройством и выведем информацию о всех loop-устройствах:

```
# losetup -fP loop-fs.img
# losetup -a
/dev/loop1: [66309]:1846788 (/root/loop-fs.img)
# ls -l /dev/loop1
brw-rw----. 1 root disk 7, 1 Feb  6 05:04 /dev/loop1
```

Создадим файловую систему в файле-образе и новую точку монтирования:

```
# mkfs -t ext4 loop-fs.img
mke2fs 1.45.5 (07-Jan-2020)
Discarding device blocks: done
Creating filesystem with 10240 1k blocks and 2560
inodes
Filesystem UUID: 8f1944c3-028a-411c-af2d-35ace61563db
```

```
Superblock backups stored on blocks:
      8193

Allocating group tables: done
Writing inode tables: done
Creating journal (1024 blocks): done
Writing superblocks and filesystem accounting infor-
mation: done

# mkdir /mnt/loop-fs
```

Выполним монтирование файловой системы с опциями по умолчанию (т.е. с опцией **-o nogrpid**):

```
# mount -o loop /dev/loop1 /mnt/loop-fs/
# mount | grep loop-fs
/dev/loop1 on /mnt/loop-fs type ext4 (rw,relatime,se-
clabel)
```

Видно, что файловая система успешно смонтирована в директории **/mnt/loop-fs/**.

Теперь перейдем к изучению вопроса принадлежности файла после его создания. Создадим в новой файловой системе тестовую директорию и файл в ней. Создадим двух новых пользователей, новую группу и включим этих пользователей в эту группу. Сделаем владельцем тестовой директории новую группу и дадим для нее все права на директорию:

```
# cd /mnt/loop-fs/
# mkdir project-2022
# useradd student1
# useradd student2
# groupadd project-2022
# usermod -aG project-2022 student1
# usermod -aG project-2022 student2
# chown :project-2022 project-2022
# chmod g+w project-2022/
# ls -ld project-2022/
drwxrwxr-x. 2 root project-2022 1024 Feb  6 05:15 pro-
ject-2022/
```

Теперь с помощью команды **su** создадим в тестовой директории файлы от имени новых пользователей:

```
# su student1
$ touch project-2022/report1
$ exit
# su student2
$ touch project-2022/report2
$ exit
# ls -l project-2022/
total 2
-rw-rw-r--. 1 student1 student1 0 Feb  6 05:24 report1
-rw-rw-r--. 1 student2 student2 0 Feb  6 05:24 report2
```

Видно, что владельцем и группой-владельцем каждого файла назначается UID и GID пользователя, создавшего файл.

Теперь установим для тестовой директории бит **set-group-ID** и повторно создадим в тестовой директории файлы от имени новых пользователей:

```
# chmod g+s project-2022/
# ls -ld project-2022/
drwxrwsr-x. 2 root project-2022 1024 Feb  6 05:24 pro-
ject-2022/
# su student1
$ touch project-2022/report12
$ exit
# su student2
$ touch project-2022/report22
$ exit
# ls -l project-2022/
total 4
-rw-rw-r--. 1 student1 student1      0 Feb   6 05:24
report1
-rw-rw-r--. 1 student1 project-2022  0 Feb   6 05:25
report12
-rw-rw-r--. 1 student2 student2      0 Feb   6 05:24
report2
-rw-rw-r--. 1 student2 project-2022  0 Feb   6 05:26
report22
```

Видно, что, как и в предыдущем случае, владельцем каждого файла назначается UID пользователя, создавшего файл, а вот группа-владелец для обоих файлов наследуется от группы-владельца директории. Таким образом, разные пользователи, входящие в одну общую группу, могут создавать файлы, доступные всем пользователям в общей группе. Это позволяет организовать совместную работу нескольких пользователей над одним проектом.

Теперь сбросим для тестовой директории бит **set-group-ID** и выполним монтирование файловой системы, указав явно опцию **-o grpid**:

```
# chmod g-s project-2022/
# ls -ld project-2022/
drwxrwxr-x. 2 root project-2022 1024 Feb  6 05:26 pro-
ject-2022/
# cd
# umount /mnt/loop-fs/
# mount -o grpid,loop /dev/loop1 /mnt/loop-fs/
```

Повторно создадим в тестовой директории файлы от имени новых пользователей:

```
# cd /mnt/loop-fs/
# su student1
$ touch project-2022/report1-grpid
$ exit
# su student2
$ touch project-2022/report2-grpid
$ exit
# ls -l project-2022/
total 6
-rw-rw-r--. 1 student1 student1       0 Feb   6 05:24
report1
-rw-rw-r--. 1 student1 project-2022  0 Feb   6 05:25
report12
-rw-rw-r--. 1 student1 project-2022  0 Feb   6 05:31
report1-grpid
-rw-rw-r--. 1 student2 student2       0 Feb   6 05:24
report2
-rw-rw-r--. 1 student2 project-2022  0 Feb   6 05:26
report22
```

```
-rw-rw-r--. 1 student2 project-2022 0 Feb  6 05:31
report2-grpid
```

Видно, что, как и в предыдущих двух случаях, владельцем каждого файла назначается UID пользователя, создавшего файл. А группа-владелец для обоих файлов наследуется от группы-владельца директории, не смотря на то, что у директории не установлен бит **set-group-ID**.

Вернем систему в исходное состояние: удалим директорию монтирования, loop-устройство и файл с образом файловой системы:

```
# cd
# umount /mnt/loop-fs/
# rmdir /mnt/loop-fs/
# losetup -d /dev/loop1
# rm loop-fs.img
```

В рассмотренном примере было показано влияние опций монтирования и наличия бита **set-group-ID** у родительской директории на то, какая группа будет у новых файлов, создаваемых в директории.

В приложении рассматривается использование интерактивного отладчика файловой системы **debugfs** для изучения индексного дескриптора файла.

Отладчик файловой системы **debugfs** позволяет исследовать и изменять состояние файловой системы **ext4**:

```
debugfs [опции] [-R запрос] [блочное_устройство]
```

Утилита **debugfs** обладает собственным набором команд и может работать как в интерактивном режиме, так и выполнять отдельные команды (опция **-R**).

Аргумент **блочное_устройство** – это имя файла блочного устройства или файла-образа, содержащего изучаемую файловую систему. Для получения содержимого индексного дескриптора файла используется подкоманда **stat**. Для указания инода его номер надо взять в угловые скобки.

Рассмотрим, как различные действия над файлом приводят к изменению значений полей структуры **ext4_inode**.

Выведем текущее время, определим блочное устройство (раздел), содержащее файловую систему, выведем идентификаторы пользователя, маску создания файлов и контекст безопасности текущей директории, создадим файл **file1** и определим его номер инода:

```
$ date
Sat 20 Nov 2021 09:33:56 PM MSK
$ df .
Filesystem     1K-blocks     Used Available Use% Mounted
on
/dev/nvme0n1p5  32765712 27192296   3879312  88% /home
$ id
uid=1000(user0) gid=1000(user0)
groups=1000(user0),10(wheel) context=unconfined_u:un-
confined_r:unconfined_t:s0-s0:c0.c1023
$ umask
0002
$ ls -dZ .
unconfined_u:object_r:user_home_t:s0 .
$ touch file1
```

```
$ ls -i file1
1854011 file1
```

Дальнейшие действия будем выполнять в режиме суперпользователя. Выведем содержимое инода с номером 1854011:

```
# debugfs -R 'stat <1854011>' /dev/nvme0n1p5
debugfs 1.45.5 (07-Jan-2020)
Inode: 1854011   Type: regular    Mode:  0664   Flags:
0x80000
Generation: 2883218480    Version: 0x00000000:00000001
User:  1000   Group:  1000   Project:     0   Size: 0
File ACL: 0
Links: 1   Blockcount: 0
Fragment:  Address: 0    Number: 0    Size: 0
 ctime: 0x61993faf:d5c84ad4 -- Sat Nov 20 21:34:23 2021
 atime: 0x61993faf:d5c84ad4 -- Sat Nov 20 21:34:23 2021
 mtime: 0x61993faf:d5c84ad4 -- Sat Nov 20 21:34:23 2021
crtime: 0x61993faf:d5c84ad4 -- Sat Nov 20 21:34:23 2021
Size of extra inode fields: 32
Extended attributes:
security.selinux     (37)     =     "unconfined_u:ob-
ject_r:user_home_t:s0\000"
Inode checksum: 0xeacd398e
EXTENTS:
```

Видно, что права доступа к файлу установлены в соответствии с пользовательской маской, идентификаторы владельца и группы унаследованы из соответствующих идентификаторов процесса, который создал файл. Все временные метки (timestamps) установлены в одинаковое значение со временем создания файла **crtime**. Контекст безопасности файла унаследован от контекста безопасности текущей директории. Цифра 37 – длина строки, содержащей контекст безопасности, включая нулевой байт. Так как файл пустой (Size: 0), то и для хранения файла не выделены ни блоки (Blockcount: 0), ни экстент (EXTENTS).

Теперь будем выполнять действия над файлом **file1** и наблюдать, как меняется содержимое полей структуры **ext4_inode**. Запишем данные в файл:

```
# cat > file1
Hello, MEPhI!
# debugfs -R 'stat <1854011>' /dev/nvme0n1p5
Inode: 1854011   Type: regular    Mode:  0664   Flags:
0x80000
Generation: 2883218480    Version: 0x00000000:00000001
User:  1000   Group:  1000   Project:     0   Size: 14
File ACL: 0
Links: 1   Blockcount: 8
Fragment:  Address: 0    Number: 0    Size: 0
 ctime: 0x6199407c:778e77f4 -- Sat Nov 20 21:37:48 2021
 atime: 0x61993faf:d5c84ad4 -- Sat Nov 20 21:34:23 2021
 mtime: 0x6199407c:778e77f4 -- Sat Nov 20 21:37:48 2021
crtime: 0x61993faf:d5c84ad4 -- Sat Nov 20 21:34:23 2021
Size of extra inode fields: 32
Extended attributes:
security.selinux     (37)     =     "unconfined_u:ob-
ject_r:user_home_t:s0\000"
Inode checksum: 0x973ae375
EXTENTS:
(0):7788915
```

В файл записано 14 байтов, включая символ перевода строки, для чего выделены блоки (Blockcount: 8) и экстент (EXTENTS:(0):7788915). Изменились время модификации самого инода **ctime** и время модификации содержимого файла **mtime**.

Выведем данные из файла:

```
# cat file1
Hello, MEPhI!
# debugfs -R 'stat <1854011>' /dev/nvme0n1p5
Inode: 1854011   Type: regular    Mode:  0664   Flags:
0x80000
Generation: 2883218480    Version: 0x00000000:00000001
User:  1000   Group:  1000   Project:     0   Size: 14
File ACL: 0
Links: 1   Blockcount: 8
Fragment:  Address: 0    Number: 0    Size: 0
 ctime: 0x6199407c:778e77f4 -- Sat Nov 20 21:37:48 2021
 atime: 0x6199472a:9093dfd8 -- Sat Nov 20 22:06:18 2021
```

```
 mtime: 0x6199407c:778e77f4 -- Sat Nov 20 21:37:48 2021
crtime: 0x61993faf:d5c84ad4 -- Sat Nov 20 21:34:23 2021
Size of extra inode fields: 32
Extended attributes:
security.selinux     (37)     =     "unconfined_u:ob-
ject_r:user_home_t:s0\000"
Inode checksum: 0xfae89f2e
EXTENTS:
(0):7788915
```

Изменилось время последнего доступа к файлу (т.е. чтения из файла) **atime**.

Создадим жесткую ссылку на файл:

```
# ln file1 file12
# debugfs -R 'stat <1854011>' /dev/nvme0n1p5
Inode: 1854011   Type: regular    Mode:  0664   Flags:
0x80000
Generation: 2883218480    Version: 0x00000000:00000001
User:  1000   Group:  1000   Project:     0   Size: 14
File ACL: 0
Links: 2   Blockcount: 8
Fragment:  Address: 0    Number: 0    Size: 0
 ctime: 0x61994951:c6f18338 -- Sat Nov 20 22:15:29 2021
 atime: 0x6199472a:9093dfd8 -- Sat Nov 20 22:06:18 2021
 mtime: 0x6199407c:778e77f4 -- Sat Nov 20 21:37:48 2021
crtime: 0x61993faf:d5c84ad4 -- Sat Nov 20 21:34:23 2021
Size of extra inode fields: 32
Extended attributes:
security.selinux     (37)     =     "unconfined_u:ob-
ject_r:user_home_t:s0\000"
Inode checksum: 0x65140425
EXTENTS:
(0):7788915
```

Увеличилось количество жестких ссылок Links и изменилось время модификации самого инода **ctime**.

Установим список прав доступа для пользователя user1 и флаг **immutable (i)**. Обратите внимание, что у файла по умолчанию уже установлен флаг **extent format (e)**.

```
# setfacl -m u:user1:rw file1
# chattr +i file1
# lsattr file1
----i---------e----- file1
# debugfs -R 'stat <1854011>' /dev/nvme0n1p5
Inode: 1854011   Type: regular    Mode:  0664   Flags:
0x80010
Generation: 2883218480    Version: 0x00000000:00000001
User:  1000   Group:  1000   Project:     0   Size: 14
File ACL: 7405608
Links: 2   Blockcount: 16
Fragment:  Address: 0    Number: 0    Size: 0
 ctime: 0x61994af0:2e11ed3c -- Sat Nov 20 22:22:24 2021
 atime: 0x6199472a:9093dfd8 -- Sat Nov 20 22:06:18 2021
 mtime: 0x6199407c:778e77f4 -- Sat Nov 20 21:37:48 2021
crtime: 0x61993faf:d5c84ad4 -- Sat Nov 20 21:34:23 2021
Size of extra inode fields: 32
Extended attributes:
security.selinux     (37)     =     "unconfined_u:ob-
ject_r:user_home_t:s0\000"
  system.posix_acl_access (28) = 01 00 00 00 01 00 06
00 02 00 06 00 e9 03 00 00
 04 00 06 00 10 00 06 00 20 00 04 00
Inode checksum: 0x028ccd21
EXTENTS:
(0):7788915
```

У файла изменилось содержимое расширенных атрибутов Extended attributes, а также изменились File ACL, флаги файла Flags и время модификации самого инода **ctime**. Обратите внимание на расширенные атрибуты: контекст безопасности хранится в пространстве **security**, а список прав доступа – в пространстве **system**.

Удалим файл. Для этого сначала снимем флаг **immutable**, а уже затем удалим обе жесткие ссылки. С установленным флагом **immutable** удалить файл не получится.

```
# chattr -i file1
# \rm file1 file12
# debugfs -R 'stat <1854011>' /dev/nvme0n1p5
Inode: 1854011   Type: regular    Mode:  0664   Flags:
0x80000
```

```
Generation: 2883218480    Version: 0x00000000:00000001
User: 1000   Group: 1000   Project:     0   Size: 0
File ACL: 0
Links: 0   Blockcount: 0
Fragment:  Address: 0    Number: 0    Size: 0
 ctime: 0x61994dfb:c922afcc -- Sat Nov 20 22:35:23 2021
 atime: 0x6199472a:9093dfd8 -- Sat Nov 20 22:06:18 2021
 mtime: 0x61994dfb:c922afcc -- Sat Nov 20 22:35:23 2021
crtime: 0x61993faf:d5c84ad4 -- Sat Nov 20 21:34:23 2021
 dtime: 0x61994dfb:(c922afcc) -- Sat Nov 20 22:35:23
2021
Size of extra inode fields: 32
Extended attributes:
security.selinux     (37)     =     "unconfined_u:ob-
ject_r:user_home_t:s0\000"
Inode checksum: 0x7e9778e5
EXTENTS:
```

Файл удалён из файловой системы, о чем свидетельствуют нулевые счётчики жёстких ссылок Links и блоков Blockcount, а также пустой экстент EXTENTS. Хотя сам файл удалён (т.е. освобождены его блоки данных, блок расширенных атрибутов и индексный дескриптор), но сам индексный дескриптор продолжает существовать в файловой системе и теперь в нем появилось время удаления файла dtime (оно, например, используется утилитами восстановления файлов).

Дмитрий Валерьевич Ефанов

Базовые механизмы защиты ядра Linux

Учебное пособие

Редактор *Е.Е. Шумакова*

Подписано в печать 18.12.2022. Формат 60x84 1/16.
Печ.л. 12,0. Уч.-изд. л. 12,0. Изд. № 023-1.

Национальный исследовательский ядерный университет «МИФИ».
115409, Москва, Каширское ш., д. 31.