



Министерство науки и высшего образования Российской Федерации  
Федеральное государственное бюджетное образовательное учреждение  
высшего образования  
«Московский государственный технический университет  
имени Н.Э. Баумана  
(национальный исследовательский университет)»  
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные  
технологии»

**РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**  
***К КУРСОВОМУ ПРОЕКТУ***  
***НА ТЕМУ:***  
***«Мониторинг системных вызовов и операций с***  
***дисками в ОС Linux»***

Студент ИУ7-75Б  
(Группа)

Кондрашова О.П.  
(Подпись, дата) (И.О.Фамилия)

Руководитель курсового проекта

Бекасов Д.Е.  
(Подпись, дата) (И.О.Фамилия)

2020 г.

**Министерство науки и высшего образования Российской Федерации  
Федеральное государственное бюджетное образовательное учреждение  
высшего образования  
«Московский государственный технический университет имени Н.Э. Баумана  
(национальный исследовательский университет)»  
(МГТУ им. Н.Э. Баумана)**

---

УТВЕРЖДАЮ  
Заведующий кафедрой \_\_\_\_\_ ИУ7  
(Индекс)  
\_\_\_\_\_ И. В. Рудаков  
(И.О.Фамилия)  
« \_\_\_\_ » \_\_\_\_\_ 2020 г.

**З А Д А Н И Е  
на выполнение курсовой работы**

по дисциплине \_\_\_\_\_ Операционные системы

Студент группы \_\_\_\_\_ ИУ7-75Б

\_\_\_\_\_ Кондрашова Ольга Павловна  
(Фамилия, имя, отчество)

Тема курсового проекта \_\_\_\_\_ Мониторинг системных вызовов и операций с дисками в ОС Linux

Направленность КП (учебный, исследовательский, практический, производственный, др.)  
\_\_\_\_\_ Учебный

Источник тематики (кафедра, предприятие, НИР) \_\_\_\_\_ Кафедра

График выполнения проекта: 25% к 4 нед., 50% к 7 нед., 75% к 11 нед., 100% к 14 нед.

**Задание** Разработать загружаемый модуль ядра для мониторинга системных вызовов и операций с дисками в операционной системе Linux. Сохранить статистику всех вызовов функций за определенные промежутки времени, и визуализировать полученные данные.

**Оформление курсового проекта:**

Расчетно-пояснительная записка на 20-30 листах формата А4.

Перечень графического (иллюстративного) материала (чертежи, плакаты, слайды и т.п.)

На защиту проекта должна быть представлена презентация, состоящая из 10-20 слайдов. На слайдах должны быть отражены: постановка задачи, использованные методы и алгоритмы, расчетные соотношения, структура комплекса программ, диаграмма классов, интерфейс, характеристики разработанного ПО.

Дата выдачи задания «12» ноября 2020 г.

**Руководитель курсового проекта**

\_\_\_\_\_ Бекасов Д.Е.  
(Подпись, дата) (И.О.Фамилия)

**Студент**

\_\_\_\_\_ Кондрашова О.П.  
(Подпись, дата) (И.О.Фамилия)

# СОДЕРЖАНИЕ

ВВЕДЕНИЕ.....	5
1. Аналитический раздел.....	6
1.1 Трассировка ядра.....	6
1.1.1 Linux Security API.....	7
1.1.2 Модификация таблицы системных вызовов .....	7
1.1.3 kprobes .....	9
1.1.4 Kernel tracepoints .....	10
1.1.5 ftrace.....	10
1.1.6 Выводы .....	12
1.2 Специальные файлы устройств .....	13
1.3 Загружаемые модули ядра .....	14
1.3.1 Пространство пользователя и пространство ядра.....	15
1.4 Визуализация.....	16
1.4.1 Loki .....	16
1.4.2 Grafana .....	16
1.5 Выводы .....	17
2. Конструкторский раздел .....	18
2.1 Общая архитектура приложения .....	18
2.2 Перехват функций .....	18
2.3 Алгоритм перехвата функции .....	19
2.4 Сбор данных для визуализации.....	21
2.5 Выводы .....	22
3. Технологический раздел .....	23
3.1 Выбор языка программирования.....	23
3.2 Инициализация ftrace .....	23

3.3	Функции-обёртки для перехватываемых функций .....	25
3.4	Примеры работы.....	26
3.5	Выводы .....	28
ЗАКЛЮЧЕНИЕ .....		29
Список литературы.....		30

## **ВВЕДЕНИЕ**

При работе с операционной системой Linux может потребоваться перехватывать вызовы важных функций внутри ядра (например, запуск процессов или запись и чтение с жесткого диска) для обеспечения возможности мониторинга активности в системе или превентивного блокирования деятельности подозрительных процессов. Курсовой проект посвящен исследованию способов перехвата вызовов функций внутри ядра с их последующим логированием и представлением в графической форме для наглядного мониторинга.

Целью данной курсовой работы является разработка загружаемого модуля ядра, позволяющего удобно перехватить любую функцию в ядре по имени и выполнить свой код вокруг её вызовов. Собранные данные о количестве вызовов функций необходимо сохранить в лог-файле и представить в графическом виде.

Для достижения поставленной цели необходимо решить следующие задачи:

- проанализировать существующие подходы для перехвата функций;
- изучить средства визуализации;
- реализовать загружаемый модуль ядра.

## **1. Аналитический раздел**

В соответствии с заданием на курсовой проект необходимо разработать программное обеспечение, перехватывающее события в системе, инициирующиеся средством исполнения экспортируемых функций ядра.

Программное обеспечение должно обеспечивать перехват всех действий нужных функций. Также необходимо предоставить возможность пользователю анализировать полученную информацию, представив собранные данные в графическом виде.

В данном разделе будут проанализированы различные подходы к трассировке ядра и перехвату функций, особенности блочных и символьных устройств, основные принципы загружаемых модулей ядра, а также технологии для визуализации.

### **1.1 Трассировка ядра**

Под трассировкой понимается получение информации о том, что происходит внутри работающей системы. Для этого используются специальные программные инструменты, регистрирующие все события в системе.

Программы-трассировщики могут одновременно отслеживать события как на уровне отдельных приложений, так и на уровне операционной системы. Полученная в ходе трассировки информация может оказаться полезной для диагностики и решения многих системных проблем.

Трассировку иногда сравнивают с логгированием. Сходство между этими двумя процедурами действительно есть, но есть и различия.

Во время трассировки записывается информация о событиях, происходящих на низком уровне. Их количество исчисляется сотнями и даже тысячами. В логи же записывается информация о высокоуровневых событиях, которые случаются гораздо реже: например, вход пользователей в систему, ошибки в работе приложений, транзакции в базах данных и другие [1].

Далее будут рассмотрены различные подходы к трассировке ядра и перехвату вызываемых функций.

### **1.1.1 Linux Security API**

Linux Security API — специальный интерфейс, созданный именно для перехвата функций. В критических местах кода ядра расположены вызовы security-функций, которые в свою очередь вызывают коллбеки, установленные security-модулем. Security-модуль может изучать контекст операции и принимать решение о её разрешении или запрете [2].

Недостатки:

- security-модули являются частью ядра и не могут быть загружены динамически;
- в системе может быть только один security-модуль (с небольшими исключениями).

Таким образом, для использования Security API необходимо поставлять собственную сборку ядра, а также интегрировать дополнительный модуль с SELinux или AppArmor, которые используются популярными дистрибутивами.

### **1.1.2 Модификация таблицы системных вызовов**

В ОС Linux все обработчики системных вызовов расположены в таблице `sys_call_table`. Подмена значений в этой таблице приводит к смене поведения всей системы. Таким образом, сохранив старые значения обработчика и подставив в таблицу собственный обработчик, можно перехватить любой системный вызов [2].

Преимущества:

- Полный контроль над любыми системными вызовами — единственным интерфейсом к ядру у пользовательских приложений;

- минимальные накладные расходы. Необходимо один раз изменить таблицу, а после этого единственным расходом является только вызов функции для вызова оригинального обработчика системного вызова;
- минимальные требования к ядру. Не требует каких-либо дополнительных конфигурационных опций в ядре, следовательно, поддерживает максимально широкий спектр систем.

Недостатки:

- Техническая сложность реализации. Сама по себе замена указателей в таблице не представляет трудностей. Но придется выполнить трудоемкие сопутствующие задачи:
  - поиск таблицы системных вызовов;
  - обход защиты от модификации таблицы;
  - атомарное и безопасное выполнение замены;
- невозможность перехвата некоторых обработчиков. В ядрах до версии 4.16 обработка системных вызовов для архитектуры x86\_64 содержала целый ряд оптимизаций. Некоторые из них требовали того, что обработчик системного вызова являлся специальным переходником, реализованным на ассемблере. Соответственно, подобные обработчики порой сложно, а иногда и вовсе невозможно заменить на свои, написанные на языке C. Более того, в разных версиях ядра используются разные оптимизации, что так же добавляет технических сложностей.
- перехватываются только системные вызовы. Этот подход позволяет заменять обработчики системных вызовов, что ограничивает точки входа только ими.

Данный подход позволяет полностью подменить таблицу системных вызовов, что является несомненным плюсом, но также ограничивает количество функций, которые можно мониторить.



### 1.1.3 kprobes

kprobes - специализированный API, в первую очередь предназначенный для отладки и трассирования ядра. Этот интерфейс позволяет устанавливать пред- и постобработчики для любой инструкции в ядре, а также обработчики на вход и возврат из функции. Обработчики получают доступ к регистрам и могут их изменять. Таким образом, можно получить как мониторинг, так и возможность влиять на дальнейший ход работы [2].

Преимущества:

- хорошо задокументированный интерфейс, работа kprobes по возможности оптимизирована;
- перехват любого места в ядре. Kprobes реализуются с помощью точек останова (инструкции `int3`), внедряемых в исполнимый код ядра. Это позволяет устанавливать kprobes в любом месте любой функции, если оно известно. Аналогично, kretprobes реализуются через подмену адреса возврата на стеке и позволяют перехватить возврат из любой функции (за исключением тех, которые управление в принципе не возвращают).

Недостатки:

- техническая сложность. Kprobes — это только способ установить точку останова в любом месте ядра. Для получения аргументов функции или значений локальных переменных надо знать, в каких регистрах или где на стеке они лежат, и самостоятельно их оттуда извлекать. Для блокировки вызова функции необходимо вручную модифицировать состояние процесса так, чтобы процессор подумал, что он уже вернул управление из функции;
- Jprobes объявлены устаревшими. Jprobes — это надстройка над kprobes, позволяющая удобно перехватывать вызовы функций. Она самостоятельно извлечёт аргументы функции из регистров или стека и вызовет обработчик, который должен иметь ту же сигнатуру, что и перехватываемая функция. Недостаток в том, что jprobes объявлены устаревшими и вырезаны из современных ядер;

- нетривиальные накладные расходы. Расстановка точек останова дорогая, но она выполняется единократно. Точки останова не влияют на остальные функции, однако их обработка относительно недешёвая. К счастью, для архитектуры x86\_64 реализована jump-оптимизация, существенно уменьшающая стоимость kprobes, но она всё ещё остаётся больше, чем, например, при модификации таблицы системных вызовов;
- kretprobes реализуются через подмену адреса возврата на стеке. Соответственно, им необходимо где-то хранить оригинальный адрес, чтобы вернуться туда после обработки kretprobe. Адреса хранятся в буфере фиксированного размера. В случае его переполнения, когда в системе выполняется слишком много одновременных вызовов перехваченной функции, kretprobes будет пропускать срабатывания;

Данный подход обладает сложной технической реализацией, а также есть вероятность возникновения ошибок после переполнения буфера памяти.

### 1.1.4 Kernel tracepoints

Kernel tracepoints — это фреймворк для трассировки ядра, сделанный через статическое инструментирование кода [3].

Преимущества:

- минимальные накладные расходы. Нужно только вызвать функцию трассировки в необходимом месте.

Недостатки:

- Отсутствие хорошо задокументированного API;
- не заработают в модуле, если включен CONFIG\_MODULE\_SIG и нет закрытого ключа для подписи.

### 1.1.5 ftrace

Ftrace — это фреймворк для трассирования ядра на уровне функций. Ftrace был разработан Стивеном Ростедтом и добавлен в ядро в 2008 году,

начиная с версии 2.6.27. Работает ftrace на базе файловой системы debugfs, которая в большинстве современных дистрибутивов Linux смонтирована по умолчанию.

Реализуется ftrace на основе ключей компилятора -pg и -mfentry, которые вставляют в начало каждой функции вызов специальной трассировочной функции mcount() или \_\_fentry\_\_(). Обычно, в пользовательских программах эта возможность компилятора используется профилировщиками, чтобы отслеживать вызовы всех функций. Ядро же использует эти функции для реализации фреймворка ftrace.

Для популярных архитектур доступна оптимизация: динамический ftrace. Суть в том, что ядро знает расположение всех вызовов mcount() или \_\_fentry\_\_() и на ранних этапах загрузки заменяет их машинный код на nop — специальную ничего не делающую инструкцию. При включении трассирования в нужные функции вызовы ftrace добавляются обратно. Таким образом, если ftrace не используется, то его влияние на систему минимально [2].

Преимущества:

- перехват любой функции по имени. Для указания интересующей функции достаточно знать только её имя;
- перехват совместим с трассировкой. Очевидно, что этот способ не конфликтует с ftrace, так что с ядра всё ещё можно снимать очень полезные показатели производительности.

Недостатки:

- требования к конфигурации ядра. Для успешного выполнения перехвата функций с помощью ftrace ядро должно предоставлять целый ряд возможностей:
  - список символов kallsyms для поиска функций по имени;
  - фреймворк ftrace в целом для выполнения трассировки;
  - опции ftrace, критически важные для перехвата.

Обычно ядра, используемые популярными дистрибутивами, все эти опции в себе всё равно содержат, так как они не влияют на производительность и полезны при отладке [3].

В таблице 1 приведен обзор рассмотренных методов.

*Таблица 1. Сравнение технологий*

	Наличие задокументированного API	Техническая простота реализации	Динамическая загрузка	Перехват всех функций	Работает в ядре с любой конфигурации
Linux Security API	-	+	-	+	-
Модификация таблицы системных вызовов	-	+	+	-	+
kprobes	+	-	+	+	+
Kernal tracepoints	-	+	+	+	-
ftrace	+	+	+	+	-

### 1.1.6 Выводы

В ходе анализа приведенных подходов к перехвату функций, был выбран фреймворк ftrace, так как он позволяет перехватить любую функцию по её имени, может быть динамически загружен в ядро, прост в реализации по сравнению с аналогами, а также имеет хорошо задокументированным API.

## 1.2 Специальные файлы устройств

В Linux все отображается в файловом виде, в том числе и устройства. Все подключённые к операционной системе Linux устройства, в том числе и жесткий диск, размещаются в каталоге `/dev/`.

В ОС Linux различают устройства блок-ориентированные и байт-ориентированные. Блок-ориентированные (или блочные) устройства, например, жесткий диск, передают данные блоками. Блочные устройства в UNIX и Linux — это устройства хранения с произвольным доступом, над которыми размещаются файловые системы. Блочное устройство обеспечивает обмен блоками данных. Блок— это единица данных фиксированного размера. Размер блока определяется ядром, но чаще всего он совпадает с размером страницы аппаратной архитектуры, и для 32-битной архитектуры x86 составляет 4096 байт. Оборудование хранит данные на физических носителях, разбитых на сектора.

Байт-ориентированные (или символьные) устройства, например, принтер и модем, передают данные посимвольно, как непрерывный поток байтов. Взаимодействие с блочными устройствами может осуществляться лишь через буферную память, а для символьных устройств буфер не требуется.

Жёсткие диски имеют особенные названия. В зависимости от интерфейса, через который подключён жёсткий диск, название может начинаться на:

- `sd` - устройство, подключённое по SCSI;
- `hd` - устройство ATA;
- `vd` - виртуальное устройство;
- `mmcblk` - обозначаются флешки, подключённые через картридер.

## 1.3 Загружаемые модули ядра

Одной из хороших особенностей Linux является способность расширения функциональности ядра во время работы. Это означает, есть возможность добавить функциональность в ядро (и убрать её), когда система запущена и работает. Часть кода, которая может быть добавлена в ядро во время работы, называется модулем. Ядро Linux предлагает поддержку довольно большого числа типов (или классов) модулей. Каждый модуль является подготовленным объектным кодом (не слинкованным для самостоятельной работы), который может быть динамически подключен в работающее ядро, а позднее может быть выгружен из ядра.

Каждый модуль ядра регистрирует себя для того, чтобы обслуживать в будущем запросы, и его функция инициализации немедленно прекращается. Иными словами, задача функции инициализации модуля заключается в подготовке функций модуля для последующего вызова. Функция выхода модуля вызывается только непосредственно перед выгрузкой модуля. Функция выхода модуля должна тщательно отменить все изменения, сделанные функцией инициализации, или функции модуля сохраняются до перезагрузки системы. Возможность выгрузить модуль помогает сократить время разработки; можно тестировать последовательные версии новых драйверов, не прибегая каждый раз к длительному циклу выключения/перезагрузки.

Модуль связан только с ядром и может вызывать только те функции, которые экспортированы ядром, нет библиотек для установления связи. Например, функция `printk`, является версией `printf`, определённой в ядре и экспортированной для модулей. Она ведёт себя аналогично оригинальной функции с небольшими отличиями [9].

### **1.3.1 Пространство пользователя и пространство ядра**

Модули работают в пространстве ядра, в то время как приложения работают в пользовательском пространстве. Это базовая концепция теории операционных систем.

На практике ролью операционной системы является обеспечение программ надёжным доступом к аппаратной части компьютера. Кроме того, операционная система должна обеспечивать независимую работу программ и защиту от несанкционированного доступа к ресурсам. Решение этих нетривиальных задач становится возможным, только если процессор обеспечивает защиту системного программного обеспечения от прикладных программ.

Выбранный подход заключается в обеспечении разных режимов работы (или уровней) в самом центральном процессоре. Уровни играют разные роли и некоторые операции на более низких уровнях не допускаются; программный код может переключить один уровень на другой только ограниченным числом способов. Unix системы разработаны для использования этой аппаратной функции с помощью двух таких уровней. Все современные процессоры имеют не менее двух уровней защиты, а некоторые, например семейство x86, имеют больше уровней; когда существует несколько уровней, используются самый высокий и самый низкий уровни. Под Unix ядро выполняется на самом высоком уровне (также называемым режимом супервизора), где разрешено всё, а приложения выполняются на самом низком уровне (так называемом пользовательском режиме), в котором процессор регулирует прямой доступ к оборудованию и несанкционированный доступ к памяти. Unix выполняет переход из пользовательского пространства в пространство ядра, когда приложение делает системный вызов или приостанавливается аппаратным прерыванием. Код ядра, выполняя системный вызов, работает в контексте процесса - он действует от имени вызывающего процесса и в состоянии получить данные в адресном пространстве процесса. Код, который обрабатывает прерывания, с другой стороны, является асинхронным по

отношению к процессам и не связан с каким-либо определённым процессом [9].

Ролью модуля является расширение функциональности ядра; код модулей выполняется в пространстве ядра.

## **1.4 Визуализация**

Визуализация количества вызовов функций нужна для того, чтобы можно было наглядно оценить состояние системы без необходимости разбираться с лог-файлами.

### **1.4.1 Loki**

Loki — это набор компонентов для полноценной системы работы с логами. В отличие от других подобных систем Loki основан на идее индексировать только метаданные логов — labels (так же, как и в Prometheus), а сами логи сжимать рядом [7].

Loki-стек состоит из трёх компонентов: Promtail, Loki, Grafana. Promtail собирает логи, обрабатывает их и отправляет в Loki. Loki их хранит. А Grafana умеет запрашивать данные из Loki и показывать их. Loki можно использовать не только для хранения логов и поиска по ним. Весь стек даёт большие возможности по обработке и анализу поступающих данных [8].

### **1.4.2 Grafana**

Grafana — это платформа с открытым исходным кодом для визуализации, мониторинга и анализа данных. Grafana позволяет пользователям создавать дашборды с панелями, каждая из которых отображает определенные показатели в течение установленного периода времени. Каждый дашборд универсален, поэтому его можно настроить для конкретного проекта или с учетом любых потребностей разработки [6].



Искать по логам можно в специальном интерфейсе Grafana — Explorer. Для запросов используется язык LogQL.

## **1.5 Выводы**

В данном разделе были проанализированы подходы к трассировке ядра и перехвату функций и выбран наиболее оптимальных метод для реализации поставленных задач. Были рассмотрены особенности блочных и символьных устройств, основные принципы загружаемых модулей ядра и понятия пространства ядра и пространства пользователя, а также приведены технологии для обеспечения визуализации данных.

## 2. Конструкторский раздел

В данном разделе будет рассмотрена общая архитектура приложения, алгоритм перехвата функций и сбор логов для визуализации.

### 2.1 Общая архитектура приложения

В состав программного обеспечения входит один загружаемый модуль ядра, который следит за вызовом нужных функций, с последующим сбором информации и визуализацией полученных данных за определенные промежутки времени.

### 2.2 Перехват функций

В листинге 1 представлена структура `struct ftrace_hook`, которая описывает каждую перехватываемую функцию.

*Листинг 1. Структура перехватываемой функции*

```
/* struct ftrace_hook описывает перехватываемую функцию
name – имя перехватываемой функции
function – адрес функции-обертки, вызываемой вместо перехваченной функции
original – указатель на мест, куда будет записан адрес перехватываемой функции
address – адрес перехватываемой функции
*/
struct ftrace_hook {
    const char *name;
    void *function;
    void *original;

    unsigned long address;
    struct ftrace_ops ops;
};
```

Пользователю необходимо заполнить только первые три поля: `name`, `function`, `original`. Остальные поля считаются деталью реализации. Описание всех перехватываемых функций можно собрать в массив и использовать макросы, чтобы повысить компактность кода. В листинге 2 представлен массив перехватываемых функций.

## Листинг 2. Массив перехватываемых функций

```
#define HOOK(_name, _function, _original) \
{ \
    .name = (_name), \
    .function = (_function), \
    .original = (_original), \
}

/* массив перехватываемых функций */
static struct ftrace_hook demo_hooks[] = {
    HOOK("__x64_sys_clone", hook_sys_clone, &orig_sys_clone),
    HOOK("__x64_sys_execve", hook_sys_execve, &orig_sys_execve),
    HOOK("bdev_read_page", hook_bdev_read_page, &orig_bdev_read_page),
    HOOK("bdev_write_page", hook_bdev_write_page, &orig_bdev_write_page),
    HOOK("random_read", hook_random_read, &orig_random_read),
};
```

## 2.3 Алгоритм перехвата функции

На рис. 1 представлена схема работы перехвата на примере функции `sys_execve()`

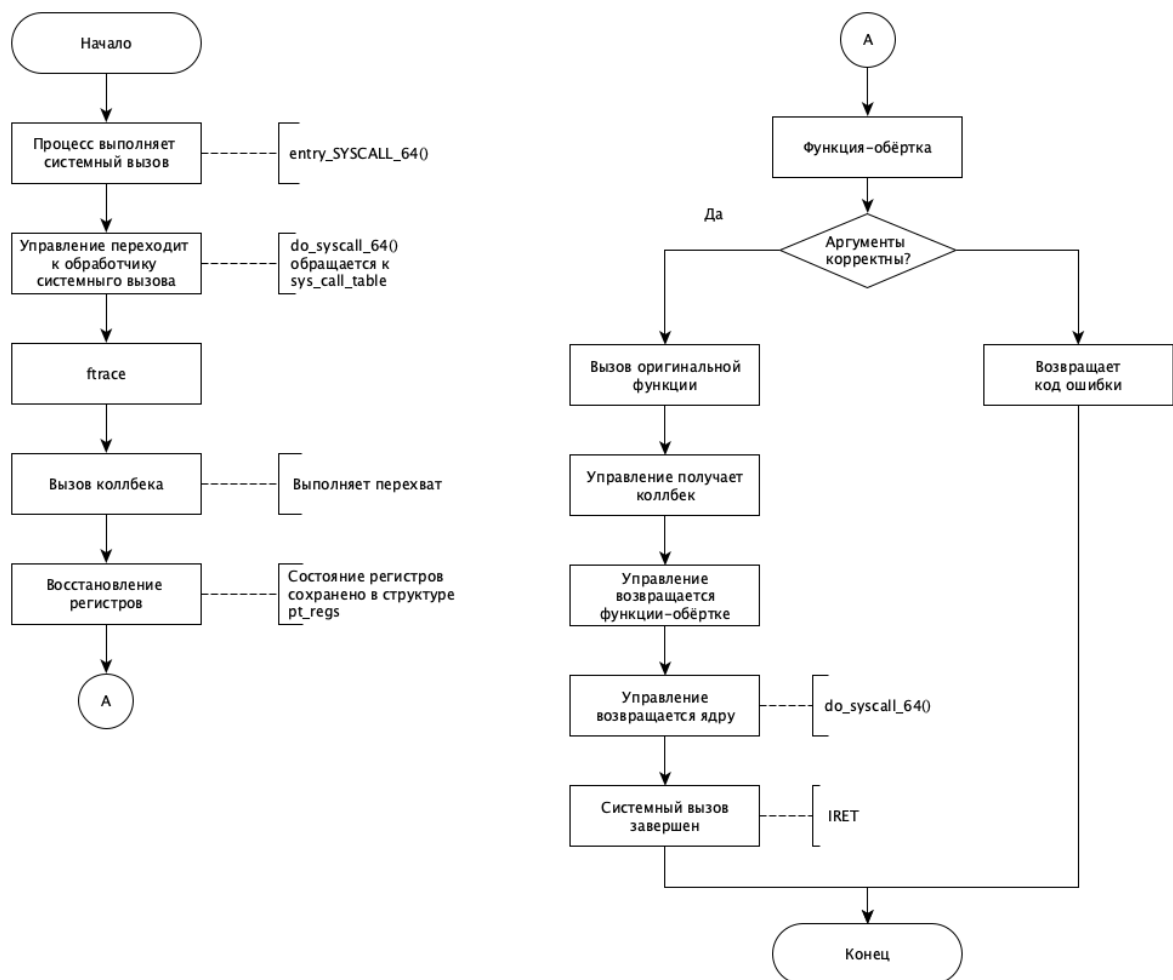


Рисунок 1. Схема перехвата функций

1. Пользовательский процесс выполняет SYSCALL. С помощью этой инструкции выполняется переход в режим ядра и управление передаётся низкоуровневому обработчику системных вызовов — `entry_SYSCALL_64()`. Он отвечает за все системные вызовы 64-битных программ на 64-битных ядрах.

2. Управление переходит к конкретному обработчику. Ядро передаёт управление высокоуровневой функции `do_syscall_64()`, написанной на C. Эта функция в свою очередь обращается к таблице обработчиков системных вызовов `sys_call_table` и вызывает оттуда конкретный обработчик по номеру системного вызова — `sys_execve()`.

3. Вызывается `ftrace`. В начале каждой функции ядра находится вызов функции `__fentry__()`, которая реализуется фреймворком `ftrace`.

4. `Ftrace` вызывает разработанный коллбек.

5. Коллбек выполняет перехват.

6. `Ftrace` восстанавливает регистры. Следуя флагу `FTRACE_SAVE_REGS`, `ftrace` сохраняет состояние регистров в структуре `pt_regs` перед вызовом обработчиков. При завершении обработки `ftrace` восстанавливает регистры из этой структуры. Наш обработчик изменяет регистр `%rip` — указатель на следующую исполняемую инструкцию — что в итоге приводит к передаче управления по новому адресу.

7. Управление получает функция-обёртка. Из-за безусловного перехода активация функции `sys_execve()` прерывается. Вместо неё управление получает функция `hook_sys_execve()`. При этом всё остальное состояние процессора и памяти остаётся без изменений, поэтому данная функция получает все аргументы оригинального обработчика и при завершении вернёт управление в функцию `do_syscall_64()`.

8. Обёртка вызывает оригинальную функцию. Функция `hook_sys_execve()` может проанализировать аргументы и контекст системного вызова (кто что запускает) и запретить или разрешить процессу его выполнение. В случае запрета функция просто возвращает код ошибки. Иначе

же ей следует вызвать оригинальный обработчик — `sys_execve()` вызывается повторно, через указатель `real_sys_execve`, который был сохранён при настройке перехвата.

9. Управление получает коллбек. Как и при первом вызове `sys_execve()`, управление опять проходит через `ptrace` и передаётся в коллбек.

10. Коллбек ничего не делает. Потому что в этот раз функция `sys_execve()` вызывается функцией `hook_sys_execve()`, а не ядром из `do_syscall_64()`. Поэтому коллбек не модифицирует регистры и выполнение функции `sys_execve()` продолжается как обычно.

11. Управление возвращается обёртке.

12. Управление возвращается ядру. Функция `hook_sys_execve()` завершается и управление переходит в `do_syscall_64()`, которая считает, что системный вызов был завершён как обычно.

13. Управление возвращается в пользовательский процесс. Наконец ядро выполняет инструкцию `IRET` (или `SYSRET`, но для `execve()` — всегда `IRET`), устанавливая регистры для нового пользовательского процесса и переводя центральный процессор в режим исполнения пользовательского кода. Системный вызов (и запуск нового процесса) завершён.

## 2.4 Сбор данных для визуализации

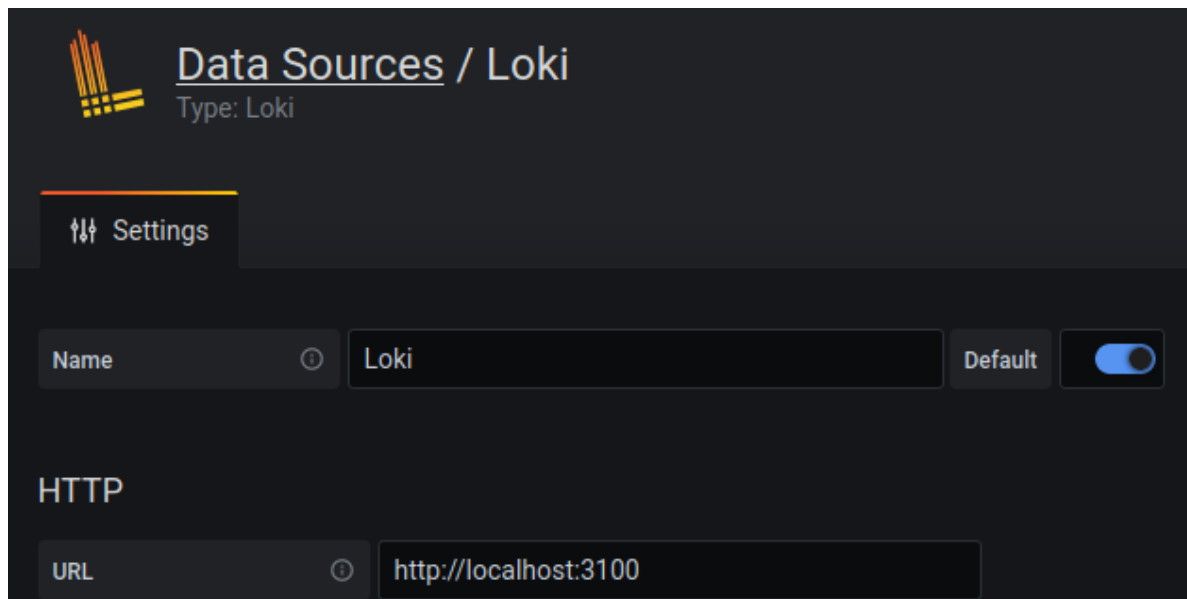
Собранные данные о вызове функций хранятся в лог-файле `/var/log/syslog`. Для того, чтобы передать собранные метрики в платформу Grafana для визуализации, необходимо настроить Promtail для считывания данных из лог-файла и их дальнейшую передачу в Loki для хранения. В листинге 3 представлен фрагмент конфигурационного файла Promtail.

*Листинг 3. Сбор метрик /var/log/syslog*

```
scrape_configs:
- job_name: system
  static_configs:
  - targets:
    - localhost
    labels:
      job: syslogs
```

```
__path__: /var/log/syslog
```

Собранные данные Loki отправляет на порт 3100. На рис. 2 представлена настройка Grafana для прослушивания Loki на порту 3100.



*Рисунок 2. Настройка Grafana*

## 2.5 Выводы

В данном разделе была рассмотрена общая архитектура приложения, алгоритм перехвата функций и сбор логов для визуализации.

### 3. Технологический раздел

В данном разделе рассматривается выбор языка программирования и реализация программного обеспечения.

#### 3.1 Выбор языка программирования

Модуль ядра написан на языке программирования C. Компилятор – gcc. Выбор языка основан на том, что исходный код системы, все модули ядра и драйверы операционной системы Linux написаны на языке C.

#### 3.2 Инициализация ftrace

Для начала требуется найти и сохранить адрес функции, которую будет перехватывать разрабатываемый модуль ядра. Ftrace позволяет трассировать функции по имени, но при этом всё равно надо знать адрес оригинальной функции, чтобы вызывать её.

Найти адрес можно с помощью kallsyms — списка всех символов в ядре. В этот список входят все символы, не только экспортируемые для модулей. Получение адреса перехватываемой функции представлено в листинге 4.

*Листинг 4. resolve\_hook\_address*

```
static int resolve_hook_address(struct ftrace_hook *hook)
{
    hook->address = kallsyms_lookup_name(hook->name);

    if (!hook->address) {
        pr_debug("unresolved symbol: %s\n", hook->name);
        return -ENOENT;
    }

    return 0;
}
```

В листинге 5 инициализируется структура ftrace\_ops. В ней обязательным полем является лишь func, указывающая на коллбек, но также необходимо установить некоторые важные флаги.

### Листинг 5. *install\_hook*

```
/* инициализация структуры ftrace_ops */
int install_hook(struct ftrace_hook *hook)
{
    int err;

    err = resolve_hook_address(hook);
    if (err)
        return err;

    hook->ops.func = ftrace_thunk;
    hook->ops.flags = FTRACE_OPS_FL_SAVE_REGS
                    | FTRACE_OPS_FL_RECURSION_SAFE
                    | FTRACE_OPS_FL_IPMODIFY;

    /* включить ftrace для функции */
    err = ftrace_set_filter_ip(&hook->ops, hook->address, 0, 0);
    if (err) {
        pr_debug("ftrace_set_filter_ip() failed: %d\n", err);
        return err;
    }

    /* разрешить ftrace вызывать коллбек */
    err = register_ftrace_function(&hook->ops);
    if (err) {
        pr_debug("register_ftrace_function() failed: %d\n", err);
        ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0);
        return err;
    }

    return 0;
}
```

В листинге 6 представлена функция отключения перехвата.

### Листинг 6. *remove\_hook*

```
void remove_hook(struct ftrace_hook *hook)
{
    int err;

    err = unregister_ftrace_function(&hook->ops);
    if (err) {
        pr_debug("unregister_ftrace_function() failed: %d\n", err);
    }

    err = ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0);
    if (err) {
        pr_debug("ftrace_set_filter_ip() failed: %d\n", err);
    }
}
```



### 3.3 Функции-обёртки для перехватываемых функций

Для перехвата необходимо определить функции-обёртки. Очень важно в точности соблюдать сигнатуру функции: порядок и типы аргументов и возвращаемого значения. Оригинальные функции были взяты из исходных кодов ядра Linux [5].

На листингах 7 и 8 представлены реализации функций-обёрток для считывания информации с символьного и блочного устройств соответственно. В качестве символьного устройства для примера взят /dev/random, блочного - /dev/sda. Функция-обёртка random\_read выводит количество считанных с устройства байт.

*Листинг 7. orig\_random\_read, hook\_random\_read*

```
static asmlinkage ssize_t (*orig_random_read)(struct file *file, char __user
*buf, size_t nbytes, loff_t *ppos);

static asmlinkage ssize_t hook_random_read(struct file *file, char __user *buf,
size_t nbytes, loff_t *ppos)
{
    int bytes_read, i;
    long error;
    char *kbuf = NULL;

    bytes_read = orig_random_read(file, buf, nbytes, ppos);
    pr_debug("random_read: read from /dev/random: %d bytes\n", bytes_read);

    kbuf = kzalloc(bytes_read, GFP_KERNEL);
    error = copy_from_user(kbuf, buf, bytes_read);

    if(error)
    {
        Pr_debug("random_read: %ld bytes could not be copied into kbuf\n",
error);
        kfree(kbuf);
        return bytes_read;
    }

    for ( i = 0 ; i < bytes_read ; i++ )
        kbuf[i] = 0x00;

    error = copy_to_user(buf, kbuf, bytes_read);
    if (error)
        pr_debug("random_read: %ld bytes could not be copied into buf\n",
error);

    kfree(kbuf);
    return bytes_read;
}
```

Листинг 8. *orig\_bdev\_read\_page, hook\_bdev\_read\_page*

```
static asmlinkage int (*orig_bdev_read_page)(struct block_device *bdev,
sector_t sector,
struct page *page);

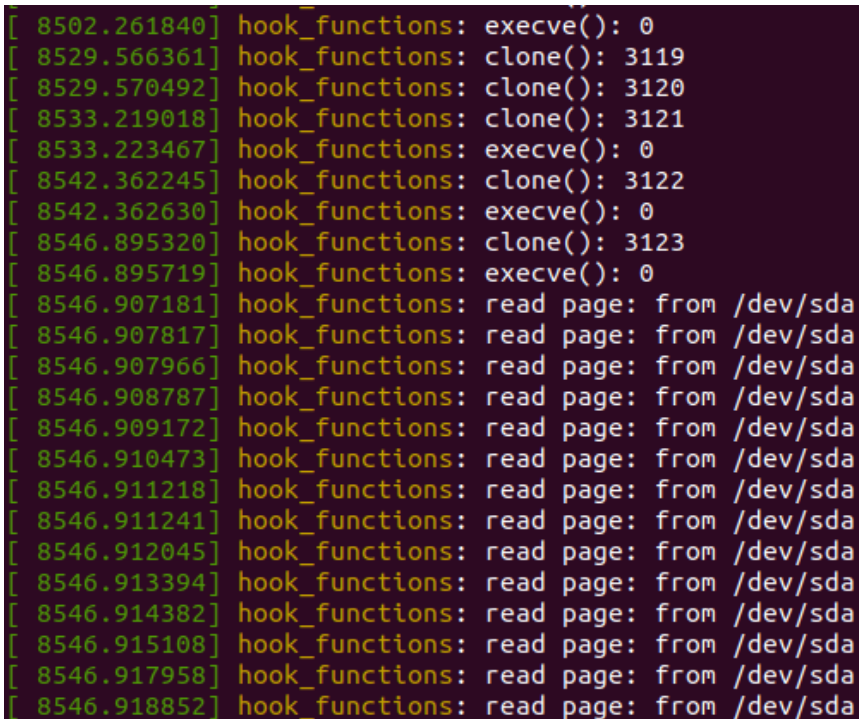
static asmlinkage int hook_bdev_read_page(struct block_device *bdev, sector_t
sector,
    struct page *page)
{
    int res;

    res = orig_bdev_read_page(bdev, sector, page);
    pr_info("read page: from /dev/sda");

    return res;
}
```

### 3.4 Примеры работы

На рис. 3 представлен пример собранных логов в /var/log/syslog.

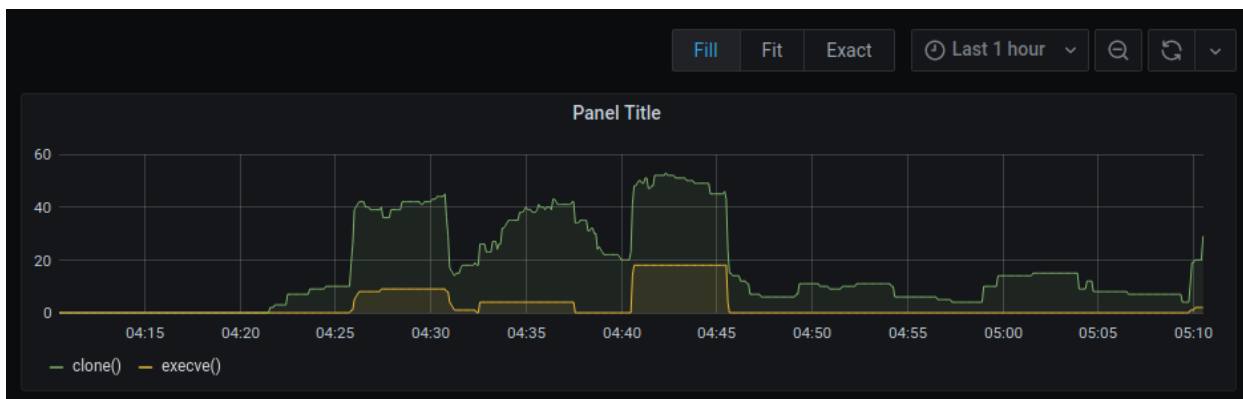


```
[ 8502.261840] hook_functions: execve(): 0
[ 8529.566361] hook_functions: clone(): 3119
[ 8529.570492] hook_functions: clone(): 3120
[ 8533.219018] hook_functions: clone(): 3121
[ 8533.223467] hook_functions: execve(): 0
[ 8542.362245] hook_functions: clone(): 3122
[ 8542.362630] hook_functions: execve(): 0
[ 8546.895320] hook_functions: clone(): 3123
[ 8546.895719] hook_functions: execve(): 0
[ 8546.907181] hook_functions: read page: from /dev/sda
[ 8546.907817] hook_functions: read page: from /dev/sda
[ 8546.907966] hook_functions: read page: from /dev/sda
[ 8546.908787] hook_functions: read page: from /dev/sda
[ 8546.909172] hook_functions: read page: from /dev/sda
[ 8546.910473] hook_functions: read page: from /dev/sda
[ 8546.911218] hook_functions: read page: from /dev/sda
[ 8546.911241] hook_functions: read page: from /dev/sda
[ 8546.912045] hook_functions: read page: from /dev/sda
[ 8546.913394] hook_functions: read page: from /dev/sda
[ 8546.914382] hook_functions: read page: from /dev/sda
[ 8546.915108] hook_functions: read page: from /dev/sda
[ 8546.917958] hook_functions: read page: from /dev/sda
[ 8546.918852] hook_functions: read page: from /dev/sda
```

Рисунок 3. */var/log/syslog*

На рис. 4 представлен результат визуализации собранных данных о системных вызовах на платформе Grafana. На графике отображены данные за последний час, сбор метрик проводился каждые 5 минут. При наведении на график можно посмотреть, сколько раз была вызвана каждая функция в конкретный момент времени.

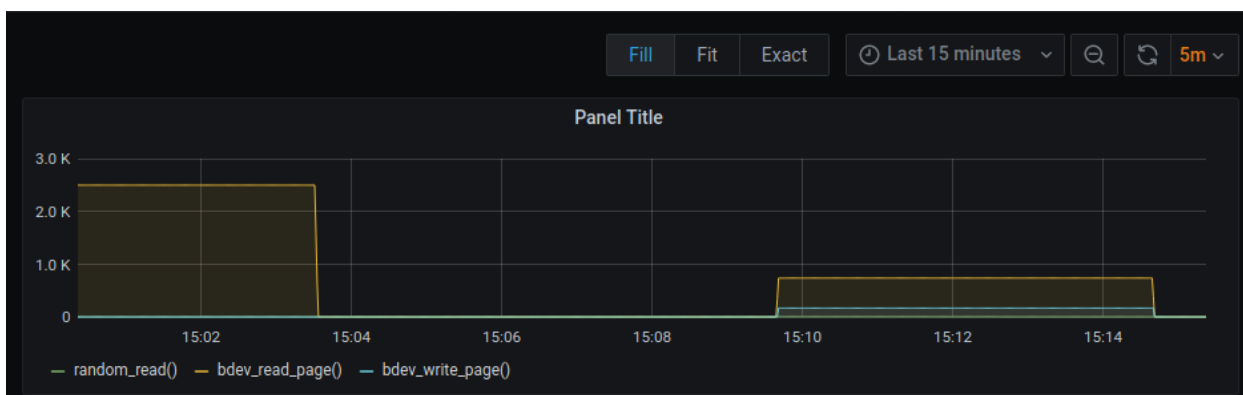
Зеленый график отображает количество вызовов `clone()`, жёлтый – `execve()`.



*Рисунок 4. Результат работы за 1 час*

На рис. 5 отображены данные об операциях с дисками за последние 15 минут, сбор метрик проводился каждые 5 минут.

Зеленый график отображает количество вызовов функции `random_read()`, жёлтый – `bdev_read_page()`, голубой – `bdev_write_page()`.



*Рисунок 5. Результат работы за 15 минут*

### **3.5 Выводы**

В данном разделе был обоснован выбор языка программирования, рассмотрены листинги реализованных функций. Приведены результаты работы ПО.

## **ЗАКЛЮЧЕНИЕ**

В ходе проделанной работы был разработан загружаемый модуль ядра, позволяющий перехватить указанные функции по их имени. В данной работе это системные вызовы для выполнения программы и создания процесса.

Проанализированы существующие подходы для перехвата функций и средства для сбора и визуализации необходимых метрик.

С помощью выбранных подходов и технологий был реализован загружаемый модуль ядра

## Список литературы

1. Руководство по трассировке системных событий в Linux [Электронный ресурс]. – Режим доступа: URL: <https://www.kernel.org/doc/Documentation/trace/events.txt> (дата обращения: 19.12.2020).
2. Перехват функций в ядре Linux с помощью ftrace [Электронный ресурс]. – Режим доступа: URL: <https://habr.com/ru/post/413241/> (Дата Обращения - 19.12.2020)
3. Трассировка ядра с ftrace [Электронный ресурс]. – Режим доступа: URL: <https://habr.com/ru/company/selectel/blog/280322/> (Дата Обращения - 19.12.2020)
4. Документация ftrace [Электронный ресурс]. – Режим доступа: URL: <https://www.kernel.org/doc/Documentation/trace/ftrace.txt> (дата обращения: 19.12.2020).
5. Исходные коды ядра Linux [Электронный ресурс]. – Режим доступа: URL: <https://github.com/torvalds/linux> (дата обращения: 19.12.2020).
6. Документация к Grafana [Электронный ресурс]. – Режим доступа: URL: <https://grafana.com/docs/grafana/latest/> (дата обращения: 19.12.2020).
7. Документация к Loki [Электронный ресурс]. – Режим доступа: URL: <https://grafana.com/docs/loki/latest/> (дата обращения: 19.12.2020).
8. Loki — сбор логов, используя подход Prometheus [Электронный ресурс]. – Режим доступа: URL: <https://habr.com/ru/company/otus/blog/487118/> (дата обращения: 19.12.2020).
9. Jessica McKellar Alessandro Rubini, Jonathan Corbet Greg Kroah-Hartman. Linux Device Drivers / Jonathan Corbet Greg Kroah-Hartman Jessica McKellar, Alessandro Rubini. — O'Reilly Media, 2016.