



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 КУРСОВОЙ РАБОТЕ НА ТЕМУ:

«Мониторинг системных вызовов в ОС Linux»

Студент группы ИУ7-75Б

(Подпись, дата)

М.А. Козлов

(И.О. Фамилия)

Руководитель курсового проекта

(Подпись, дата)

Н. Ю. Рязанова

(И.О. Фамилия)

2021 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)
« ____ » _____ 2021 г.

ЗАДАНИЕ
на выполнение курсового проекта

по дисциплине Операционные системы

Студент группы ИУ7-75Б

Козлов Максим Антонович
(Фамилия, имя, отчество)

Тема курсового проекта Мониторинг системных вызовов в ОС Linux.

Направленность КП (учебный, исследовательский, практический, производственный)
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.

Техническое задание Разработать загружаемый модуль ядра, который обладает следующим функционалом – вывод информации о вызове и передаваемых параметрах функций: bdev_read_page, bdev_write_page, sys_read, sys_write, sys_open, sys_close, random_read.

Оформление курсового проекта:

Расчетно-пояснительная записка на 25-30 листах формата А4.

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

На защиту проекта должна быть представлена презентация, состоящая из 15-20 слайдов. На слайдах должны быть отражены: постановка задачи, использованные методы и алгоритмы, расчетные соотношения, структура комплекса программ, диаграмма классов, интерфейс, характеристики разработанного ПО, результаты проведенных исследований.

Дата выдачи задания « ____ » _____ 2021г.

Руководитель курсовой работы

(Подпись, дата)

Н. Ю. Рязанова

(И.О.Фамилия)

Студент

(Подпись, дата)

М. А. Козлов

(И.О.Фамилия)

СОДЕРЖАНИЕ

Введение	4
1 Аналитическая часть	5
1.1 Существующие решения для трассировки ядра	5
1.1.1 Linux Security API	5
1.1.2 Модификация таблицы системных вызовов	5
1.1.3 kprobes	7
1.1.4 Kernel tracepoints	8
1.1.5 Сплайсинг	8
1.1.6 ftrace	9
1.1.7 Вывод	10
1.2 Загружаемые модули ядра Linux	10
1.3 Пространство пользователя и пространство ядра	10
1.4 Вывод	10
2 Конструкторская часть	11
2.1 Общая архитектура приложения	11
2.2 Перехват функций	11
2.2.1 Перехват функций через системную таблицу	11
2.2.2 Перехват функций через ftrace	11
2.3 Связь структур	11
2.4 Вывод	11
3 Технологическая часть	12
3.1 Выбор языка программирования	12
3.2 Модификация таблицы системных вызовов	12
3.3 Функции-обёртки перехватываемых системных вызовов	12
3.4 Инициализация ftrace	12
3.5 Функции-обёртки перехватываемых ftrace функций	12
3.6 Примеры работы	12
3.7 Вывод	12
4 Исследовательская часть	13
Заключение	14
Список использованных источников	15

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.

Целью данной курсовой работы является разработка загружаемого модуля ядра, позволяющего логировать вызов следующих функций: `bdev_read_page`, `bdev_write_page`, `sys_read`, `sys_write`, `sys_open`, `sys_close`, `random_read`.

Для достижения поставленной цели требуется решить следующие задачи:

- 1) проанализировать существующие подходы для перехвата функций;
- 2) реализовать загружаемый модуль ядра;
- 3) исследовать поведение перехваченных функций.

1 Аналитическая часть

В данном разделе будут проанализированы различные подходы к трассировке ядра и перехвату функций, а также основные принципы загружаемых модулей ядра.

1.1 Существующие решения для трассировки ядра

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

1.1.1 Linux Security API

Наиболее правильным было бы использование Linux Security API — специального интерфейса, созданного именно для этих целей. В критических местах ядерного кода 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.

К сожалению, у Linux Security API есть пара важных ограничений:

- 1) security-модули не могут быть загружены динамически, являются частью ядра и требуют его пересборки
- 2) в системе может быть только один security-модуль (с небольшими исключениями)

Если по поводу множественности модулей позиция разработчиков ядра неоднозначная, то запрет на динамическую загрузку принципиальный: security-модуль должен быть частью ядра, чтобы обеспечивать безопасность постоянно, с момента загрузки. 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

Мониторинг требовался в основном для действий, выполняемых пользовательскими приложениями, так что в принципе мог бы быть реализован на уровне системных вызовов. Как известно, Linux хранит все обработчики системных вызовов 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, мы можем перехватить любой системный вызов.

У этого подхода есть определённые преимущества:

— Полный контроль над любыми системными вызовами — единственным интерфейсом к ядру у пользовательских приложений. Используя его мы можем быть уверены, что не пропустим какое-нибудь важное действие, выполняемое пользовательским процессом.

— Минимальные накладные расходы. Есть единоразовые капитальные вложения при обновлении таблицы системных вызовов. Помимо неизбежной полезной нагрузки мониторинга, единственным расходом является лишний вызов функции (для вызова оригинального обработчика системного вызова).

— Минимальные требования к ядру. При желании этот подход не требует каких-либо дополнительных конфигурационных опций в ядре, так что в теории поддерживает максимально широкий спектр систем.

Однако, подход не лишен недостатков:

— Техническая сложность реализации. Сама по себе замена указателей в таблице не представляет трудностей. Но сопутствующие задачи требуют неочевидных решений и определённой квалификации:

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 переходничком, реализованным на ассемблере. Соответственно, подобные обработчики порой сложно, а иногда и вовсе невозможно заменить на свои, написанные на Си. Более того, в разных версиях ядра используются разные оптимизации, что добавляет в копилку технических сложностей.

— Перехватываются только системные вызовы. Этот подход позволяет заменять обработчики системных вызовов, что ограничивает точки входа только ими. Все дополнительные проверки выполняются либо в начале, либо в конце, и у нас есть лишь аргументы системного вызова и его возвращаемое значение. Иногда это приводит к необходимости дублировать проверки на адекватность аргументов и проверки доступа. Иногда вызывает лишние накладные расходы, когда требуется дважды копировать память пользовательского процесса: если аргумент передаётся через указатель, то его сначала придётся скопировать нам самим, затем оригинальный обработчик скопирует аргумент ещё раз для себя. Кроме того, в некоторых случаях системные вызовы предоставляют слишком низкую гранулярность событий, которые приходится дополнительно фильтровать от шума.

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

1.1.3 kprobes

Одним из вариантов, которые рассматривались, было использование kprobes: специализированного API, в первую очередь предназначенного 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 было бы получить как мониторинг, так и возможность влиять на дальнейший ход работы.

Преимущества, которые даёт использование kprobes для перехвата:

- Зрелый API. Kprobes существуют и улучшаются с 2002 года. Они обладают хорошо задокументированным интерфейсом, большинство подводных камней уже найдено, их работа 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 буквально 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 kprobes:

- 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 ваш обработчик, который должен иметь ту же сигнатуру, что и перехватываемая функция. Подвох 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 `jmp`-оптимизация, существенно уменьшающая стоимость kprobes, но она всё ещё остаётся больше, чем, например, при модификации таблицы системных вызовов.

- Ограничения kretprobes. 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.

— Отключенное вытеснение. Так как kprobes основывается на прерываниях и жонглирует регистрами процессора, то для синхронизации все обработчики выполняются с отключенным вытеснением (preemption). Это накладывает определённые ограничения на обработчики: в них нельзя ждать — выделять много памяти, заниматься вводом-выводом, спать в таймерах и семафорах, и прочие известные вещи.

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 Сплайсинг

Классический способ перехвата функций, заключающийся в замене инструкций в начале функции на безусловный переход, ведущий в обработчик. Оригинальные инструкции переносятся в другое место и исполняются перед переходом обратно в перехваченную функцию. С помощью двух переходов вшивается (splice in) свой дополнительный код в функцию, поэтому такой подход называется сплайсингом.

Именно таким образом и реализуется jump-оптимизация для kprobes. Используя сплайсинг можно добиться тех же результатов, но без дополнительных расходов на kprobes и с полным контролем ситуации.

Преимущества сплайсинга:

— Минимальные требования к ядру. Сплайсинг не требует каких-либо особенных опций в ядре и работает в начале любой функции. Нужно только знать её адрес.

— Минимальные накладные расходы. Два безусловных перехода — вот и все действия, которые надо выполнить перехваченному коду, чтобы передать управление обработчику и обратно. Подобные переходы отлично предсказываются процессором и являются очень дешёвыми.

Недостатки:

— Техническая сложность. Она зашкаливает. Нельзя просто так взять и переписать машинный код. Вот краткий и неполный список задач, которые придётся решить:

синхронизация установки и снятия перехвата (что если функцию вызовут прямо в процессе замены её инструкций?)

обход защиты на модификацию регионов памяти с кодом

инвалидация кешей процессора после замены инструкций

дизассемблирование заменяемых инструкций, чтобы скопировать их целиком

проверка на отсутствие переходов внутри заменяемого куска

проверка на возможность переместить заменяемый кусок в другое место

1.1.6 ftrace

Ftrace — это фреймворк для трассирования ядра на уровне функций. Он разрабатывается с 2008 года и обладает удобным интерфейсом для пользовательских программ. Ftrace позволяет отслеживать частоту и длительность вызовов функций, отображать графы вызовов, фильтровать интересующие функции по шаблонам, и так далее.

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.

Вызывать ftrace из каждой функции — это, разумеется, не дёшево, поэтому 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 пор — 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.

Преимущества:

- Зрелый API и простой код. Использование готовых интерфейсов в ядре существенно упрощает код. Вся установка перехвата требует пары вызовов функций, заполнение двух полей в структуре. Остальной код — это исключительно бизнес-логика, выполняемая вокруг перехваченной функции.

- Перехват любой функции по имени. Для указания интересующей нас функции достаточно написать её имя в обычной строке. Не требуются какие-то особые реверансы с редактором связей, разбор внутренних структур данных ядра, сканирование памяти, или что-то подобное. Можно перехватить любую функцию (даже не экспортируемую для модулей), зная лишь 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. Использование `kprobes` или сплайсинга может помешать механизмам ftrace.

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

Все эти возможности не являются критичными для функционирования системы и могут быть отключены в конфигурации ядра. Правда, обычно ядра, используемые популярными дистрибутивами, все эти опции в себе всё равно содержат, так как они не влияют на производительность и полезны при отладке. Однако, если вам необходимо поддерживать какие-то особенные ядра, то следует иметь в виду эти требования.

— Накладные расходы на `ftrace` меньше, чем у `kprobes` (так как `ftrace` не использует точки останова), но они выше, чем у сплайсинга, сделанного вручную. Действительно, динамический `ftrace`, является сплайсингом, только вдобавок выполняющий код `ftrace` и другие коллбеки.

— Оборачиваются функции целиком. Как и традиционный сплайсинг, данный подход полностью оборачивает вызовы функций. Однако, если сплайсинг технически возможно выполнить в любом месте функции, то `ftrace` срабатывает исключительно при входе. Естественно, обычно это не вызывает сложностей и даже наоборот удобно, но подобное ограничение иногда может быть недостатком.

1.1.7 Вывод

В таблице 1 приведен обзор рассмотренных методов.

В ходе анализа приведенных подходов к перехвату функций, был выбран сист таблицы и 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 обладает хорошо задокументированным API.

1.2 Загружаемые модули ядра Linux

1.3 Пространство пользователя и пространство ядра

1.4 Вывод

В данном разделе были проанализированы подходы к трассировке ядра и перехвату функций и выбран [todo] наиболее оптимальных метод для реализации поставленных задач. Были рассмотрены основные принципы загружаемых модулей ядра и понятия пространства ядра и пространства пользователя.

2 Конструкторская часть

В данном разделе будет рассмотрена общая архитектура приложения и алгоритм перехвата функций с помощью системной таблицы и ftrace.

2.1 Общая архитектура приложения

2.2 Перехват функций

2.2.1 Перехват функций через системную таблицу

описание, код, алгоритм

2.2.2 Перехват функций через ftrace

описание, код, алгоритм

2.3 Связь структур

Получение по файловому дескриптору имени файла.

2.4 Вывод

В данном разделе была рассмотрена общая архитектура приложения, алгоритм перехвата функций и способ получения имени файла по файловому дескриптору.

3 Технологическая часть

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

3.1 Выбор языка программирования

Операционная система Linux позволяет писать загружаемые модули ядра на Rust и на C. Для реализации загружаемого модуля был выбран последний, так как большая часть ядра и загружаемых моделей написана на языке C, а также у меня есть опыт разработки модулей на данном языке программирования.

3.2 Модификация таблицы системных вызовов

3.3 Функции-обёртки перехватываемых системных вызовов

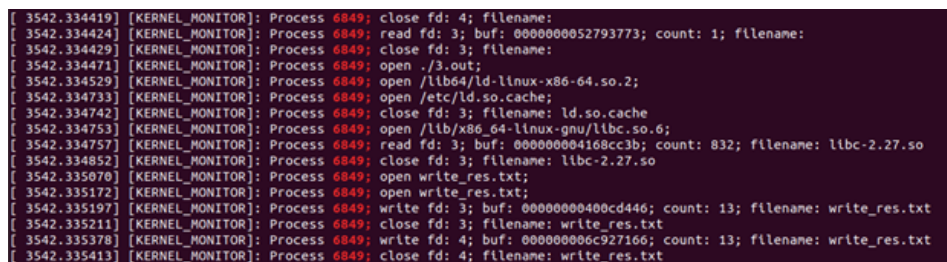
3.4 Инициализация ftrace

3.5 Функции-обёртки перехватываемых ftrace функций

3.6 Примеры работы

TODO обновить скрин.

На рисунке 3.1 представлен пример собранных логов в /var/log/syslog.



```
[ 3542.334419] [KERNEL_MONITOR]: Process 6849; close fd: 4; filename:
[ 3542.334424] [KERNEL_MONITOR]: Process 6849; read fd: 3; buf: 000000052793773; count: 1; filename:
[ 3542.334429] [KERNEL_MONITOR]: Process 6849; close fd: 3; filename:
[ 3542.334471] [KERNEL_MONITOR]: Process 6849; open ./3.out;
[ 3542.334529] [KERNEL_MONITOR]: Process 6849; open /lib64/ld-linux-x86-64.so.2;
[ 3542.334742] [KERNEL_MONITOR]: Process 6849; open /etc/ld.so.cache;
[ 3542.334733] [KERNEL_MONITOR]: Process 6849; close fd: 3; filename: ld.so.cache
[ 3542.334753] [KERNEL_MONITOR]: Process 6849; open /lib/x86_64-linux-gnu/libc.so.6;
[ 3542.334757] [KERNEL_MONITOR]: Process 6849; read fd: 3; buf: 000000004168cc3b; count: 832; filename: libc-2.27.so
[ 3542.334852] [KERNEL_MONITOR]: Process 6849; close fd: 3; filename: libc-2.27.so
[ 3542.335070] [KERNEL_MONITOR]: Process 6849; open write_res.txt;
[ 3542.335172] [KERNEL_MONITOR]: Process 6849; open write_res.txt;
[ 3542.335197] [KERNEL_MONITOR]: Process 6849; write fd: 3; buf: 00000000400cd446; count: 13; filename: write_res.txt
[ 3542.335211] [KERNEL_MONITOR]: Process 6849; close fd: 3; filename: write_res.txt
[ 3542.335378] [KERNEL_MONITOR]: Process 6849; write fd: 4; buf: 00000000c927166; count: 13; filename: write_res.txt
[ 3542.335413] [KERNEL_MONITOR]: Process 6849; close fd: 4; filename: write_res.txt
```

Рисунок 3.1 — Пример работы загружаемого модуля ядра.

3.7 Вывод

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

4 Исследовательская часть

В данном разделе

Заключение

В данной работе был реализован загружаемый модуль ядра операционной системы Linux. В процессе разработки был реализован подход, позволяющий перехватить необходимые функции и системные вызовы, и логировать необходимую информацию без перехода в режим пользователя. Реализуемый модуль поддерживает ядра версий 5.0 для архитектуры x86_64

Список использованных источников