Министерство образования и науки Российской Федерации

Федеральное государственное автономное образовательное учреждение
высшего профессионального образования
«Московский физико-технический институт
(государственный университет)»

Факультет управления и прикладной математики

Кафедра теоретической и прикладной информатики

ИССЛЕДОВАНИЕ ВОЗМОЖНОСТИ ИСПОЛЬЗОВАНИЯ КОДА ЯДРА LINUX В ПОЛЬЗОВАТЕЛЬСКИХ ПРИЛОЖЕНИЯХ

Выпускная квалификационная работа
(бакалаврская работа)

Направление подготовки:010900 Прикладные математика и физика

Выполнил:
студент 276 группы ___________________ Дуплякин Владислав Олегович

Научный руководитель:
к.ф.-м.н. ___________________ Емельянов Павел Владимирович

Москва 2016

Оглавление

1. Введение……………………………………………………………………...4

1.1 Постановка задачи……………………………………………………4

1.2 Цель работы……………...………………………...……………………...4

1.3 Обзор существующих методов решения…….……….………………..4

1.3.1 Linux Kernel Library(LKL)..……...…………………………………4

2. Изучение программного комплекса LKL……………………………......…6

2.1 Общий обзор LKL………………………………………………………6

2.1.1 Цели LKL…………………………………………………………....6

2.1.2 Общая схема LKL…..…………....……………………………………7

2.1.3 Схема взаимодействия в LKL.......………………………………....7

2.2 Управление памятью в LKL………………………………………….8

2.2.1 Анализ организации подсистемы виртуальной памяти ……......9

2.2.2 Исследование статистики /proc…………………………………10

2.2.3 Исследование с помощью отладчика…………………………...11

2.3 Поддержка потоков………..…………..……………………………...11

2.4 Управление потоками…………………...……………………………13

2.5 Системные вызовы …….……………..………………………………16

2.5.1 Общая информация о системных вызовах……………………..16

2.5.2 Системные вызовы в LKL……………………………………..19

2.5.3 Использование системных вызовов в LKL……….……………21

3. Реализация…………………..……………………………………………25

3.1 Формулировка задачи в терминах LKL………………………………….25

3.2 Блочные устройства в LKL….……………………………………………...26

3.3 Исследование возможности работы с файловыми системами в LKL…27

3.4 Интегрирование логирующей функции в LKL…………………………28

3.5 Нагрузка файловой системы…………………………………………...31

3.6 Сбор статистики и анализ получившейся информации………………...31

3.7 Полученная статистика…………………………………………………...34

4. Заключение……………………………………………………………….35
4.1 Результаты………………………………………………………….35
4.2 Перспективы развития....…………………………………………..…35
4.3 Возможное применение…..……………………………………….35
5. Список литературы…………………………………………………………36

1. Введение

Данная работа является попыткой исследования дополнительных возможностей тестирования ядра ОС Linux. В ядре Linux используется огромное количество стандартных алгоритмов и эвристических моделей. Для проверки их качества и надёжности обычно используются тесты, требующие загрузку ядра на физическом сервере или в виртуальной машине. Однако, определённый набор используемых ядром алгоритмов может быть протестирован без стандартной загрузки самого ядра.

1.1 Постановка задачи

Изучить возможность загрузки ядра Linux в контекст пользовательского приложения и безопасной передачи управления на требуемые участки кода. Исследовать шаблоны нагрузки на файловые системы. Разные программы по-разному используют файловые системы. Хотелось бы иметь возможность собирать статистическую информацию об этом использовании. Собранная статистика должна быть такой, чтобы по ней можно было пытаться "воспроизводить" нагрузку.

1.2 Цель работы

Исследование шаблонов нагрузки на файловые системы проводится с той целью, чтобы увеличить эффективность работы с файловыми системами. В файловых системах самое проблемное место это аллокатор блоков. Если аллокатор блоков работает плохо, то при работе с файловой системой получается много произвольных запросов к диску. Если аллокатор блоков работает хорошо, то запросы к диску в основном последовательные. Статистика должна давать достаточно информации, чтобы понимать хорошо или плохо работает аллокатор блоков.

1.3 Обзор существующих методов решения

1.3.1 Linux Kernel Library (LKL)

Ядро Linux является одним из самых больших проектов программного обеспечения (общее количество строк кода ядра в версии 3.18.9 превысило 15

миллионов, а всего файлов в репозитории порядка 37 тысяч [4]). Благодаря своей модели развития (open source, большое количество разработчиков и этапы review), код ядра остается высокого качества даже для самых сложных компонентов (например, драйверы файловых систем, поддерживающих ядром). Соответственно, в силу надежности некоторые части Linux могут быть использованы вне ядра, например, в утилите для чтения таких файловых систем, как ext3 или ext4.

Однако ядро Linux – достаточно взаимосвязанный код. Если даже суметь выделить какую-то часть и успешно использовать вне ядра, то потом придется отслеживать, не добавлены ли новые исправления или дополнительные улучшения в эту часть в основной ветке разработки Linux. И, возможно, новые добавляемые особенности потребуют больших усилий, чтобы они были так же добавлены в эту выделенную и используемую вне ядра часть. Более того, обратное тоже верно: исправления обнаруженных ошибок в части, используемой вне ядра, возможно, не так легко будет предложить сообществу разработчиков Linux.

Linux Kernel Library (**LKL**) – это проект, который организует код ядра Linux в такой форме, чтобы можно было его использовать обычными приложениями. С использованием LKL, усилия, потраченные на получение функциональности кода Linux вне ядра, ограничиваются компилированием кода ядра (включая патчи LKL) и связыванием приложения, которое собирается использовать код ядра, с получившейся библиотекой.

LKL позволяет использовать такие подсистемы ядра Linux, как виртуальная файловая система , сетевой стек, планировщик [1]. Однако в рамках данной работы рассматривается возможность использования драйверов файловых систем, поддерживаемых Linux. Более того, LKL может использоваться в разных средах (Linux, Windows), если данная среда предоставит некоторые примитивы для данной библиотеки.

2. Изучение программного комплекса LKL

2.1 Общий обзор LKL

2.1.1 Цели LKL

Главная цель LKL – обеспечить простой способ использования кода ядра Linux для приложений в различных средах. Авторы этого проекта преследуют следующие цели [1]:

- LKL не должен требовать определенной конкретной операционной системы (environment), на которой будет использоваться эта библиотека;
- Эта библиотека может использоваться, как в пространстве пользователя (user space), так и в пространстве ядра (kernel space);
- Репозиторий LKL должен легко обновляться из основной ветки разработки Linux;
- Требовать минимум кода от самого приложения, которое хочет использовать эту библиотеку;
- Предоставлять стабильный и легко используемый API.

Чтобы LKL мог легко обновляться из главной ветки Linux, разработчики LKL требуют механизм полного отделения LKL-специфических компонент от кода ядра.

LKL не должен зависеть от платформы, поэтому данная библиотека реализована как виртуальная компьютерная архитектура *lkl*, соответственно, в ней нет никакого платформо-зависимого кода. Вместо этого приложение, использующее LKL, должно предоставить библиотеке реализацию небольшого набора платформо-зависимых примитивов[1].

Чтобы взаимодействовать с ядром, приложение использует интерфейс, предоставляемый LKL и основанный на системных вызовах Linux.

---

[1] Эти примитивы называются *native operations* [1]

2.1.2 Общая схема LKL

Библиотека LKL взаимодействует с приложением через интерфейс, который включает в себя:

- LKL-специфичные операции[2];
- операции для данного окружения (предоставляемые приложением);
- API, похожий на API прерываний (*interrupt-like API*), который позволяет приложению уведомлять ядро Linux о внешних событиях.

2.1.3 Схема взаимодействия в LKL

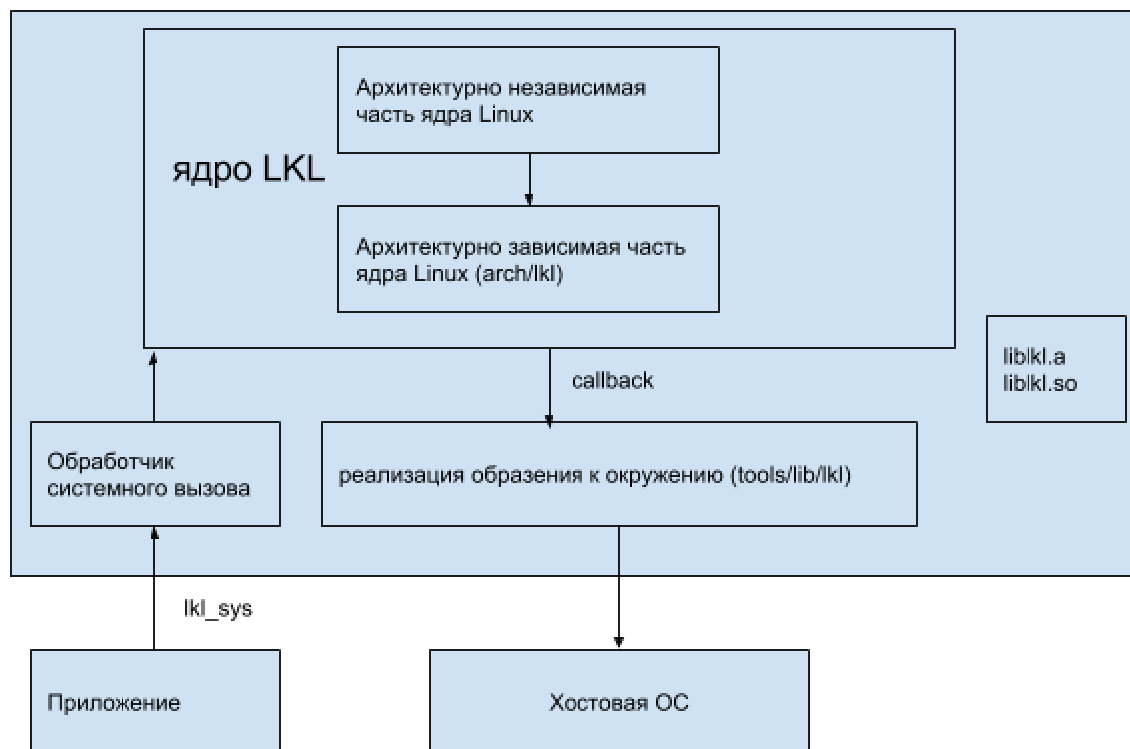

Рис. 1. Общая схема LKL.

Реализация связи между ядром LKL и окружением осуществляется следующим образом:

1. Архитектурно-независимая часть ядра Linux (стандартное ядро, собранное с нужной для LKL конфигурацией)

---

[2] Авторы называют их *LKL system calls* [1], намекая на то, что их интерфейс основан на системных вызовах Linux

2. Архитектурно-зависимая часть ядра (linux/arch/lkl)
3. Модуль реализации функций связи с окружением и внешними приложениями (linux/tools/lkl/lib). Эта часть не является частью ядра LKL.

Для выполнения запросов к внешнему миру (т.е. операции, которые не предоставляет оборудование, и которые должны быть реализованы программно) архитектурно-зависимая часть ядра осуществляет с помощью вызовов из таблицы lkl_host_ops. Также функции обратного вызова (callback) используются в драйвере виртуального устройства.

Модуль реализации функций связи с окружением и внешними приложениями:

1. предоставляет для ядра таблицу вызовов lkl_host_ops (выделение памяти, семафоров, блокировок, потоков и т.д.)
2. Этот же модуль предоставляет пользовательскому процессу возможность добавить блочные виртуальные устройства в ядро, запуск "псевдосистемных вызовов" (lkl_sys_…), а также функции запуска и остановки ядра.

Указанные компоненты и составляют указанную на рис.1 библиотеку liblkl.

В рамках исследования различных подсистем LKL рассмотрим следующие важные компоненты.

2.2 Управление памятью

В ходе исследования был проведен анализ взаимодействия lkl с памятью, а именно: последовательность действий, совершаемая lkl при организации доступа к памяти (get_free_page()). Был проведен как статический (исследование исходного кода lkl), так и динамический анализ поведения данной системы. В качестве динамического анализа была проведена трассировка утилиты, использующей большую часть функционала lkl, а именно: usermode драйвера

файловых систем lklfuse. Кроме того, было был проведен анализ использования памяти данным приложением с помощью системной иерархии /proc.

2.2.1 Анализ организации подсистемы виртуальной памяти в LKL

Некоторые подсистемы нуждаются в поддержке виртуальной памяти. Преимуществом ядра Linux является то, что оно умеет работать с виртуальной памятью даже на архитектурах на которых нет поддержки MMU. Если нет аппаратной поддержки, то Linux производит эту поддержку на программном уровне. Это означает, что все вызовы идут через внутреннюю подсистему управления памятью, которая выполняет работу по разделению адресных пространств процессов. В LKL, являющимся непривелигированным процессом пользовательского уровня, используется именно механизм программной поддержки управления памятью. Для этого в конфигурационном файле устанавливается флаг !MMU - декларирующий то, что LKL - это архитектура без поддержки MMU. Кроме того, это означает, что при данной архитектуре сборки ядро Linux “полагает”, что мы находимся в незащищенном режиме, что, в свою очередь, предполагает то, что отсутствует разграничение доступа к данным ядра и (отсутствующих в данной архитектуре) пользовательских процессов.

Ядро Linux в LKL получает на входе блок памяти (общий для всех потоков ядра) и работает с ним подобно тому, как работает любое приложение в незащищённом режиме.
Поскольку LKL работает в незащищённом режиме, то у кода ядра внутри LKL нет привилегированного режима(при переходе из user space в kernel space не происходит смена привилегий)
Так как LKL не требует механизмов защиты памяти, то память устроена просто: LKL просто нужен кусок физической памяти, который ядро Linux использует для своей работы. Из этого куска памяти lkl берёт для себя страницы точно так же, как обычное ядро Linux. Этот кусок памяти управляется ядром используя смесь buddy, SLUB/SLAB/SLOB/SLQB алгоритмов. Никакой специфики, связанной с

получением страниц в LKL нет. Так как LKL сконфигурирован так, что виртуализация памяти отсутствует, то, как следствие, отсутствует подгрузка/выгрузка страниц и своппинг.

2.2.2 Исследование статистики /proc

Иерархия /proc показывает детальную информацию о состоянии процесса операционной системы, в том числе: количество отведенной памяти, число открытых файловых дескрипторов и их назначение, и т.д. В частности, в файле /proc/[идентификатор процесса]/maps лежит информация о распределении памяти. В нашем случае для процесса в котором есть lkl, появляется vma размером с выделенный lkl-ю ram-ом.

В /proc/`pidof lklfuse`/maps можно найти строку следующего вида:

72.003906 7fe3b37ff000-7fe3b8000000 rwxp 00000000 00:00 0
[stack:29402]

Самое первое число обозначает размер памяти в мегабайтах, выделенной ядром LKL под исполнение процесса lklfuse. В данном случае количество выделяемой памяти под процесс равно 72Мб. В программе можно найти строчку, в которой резервируется этот объём памяти:

ret = lkl_start_kernel(&lkl_host_ops, lklfuse.mb * 1024 * 1024, "");

При этом по умолчанию поле mb в структуре lklfuse равно 64.

Оставшая память используется для кода программы, статических переменных и т.д. При этом если изменить поле структуры lklfuse: mb = 220, то в в /proc/`pidof lklfuse`/maps можно найти строку :

228.003906 7f9a49bff000-7f9a58000000 rwxp 00000000 00:00 0
[stack:16354]

Таким образом, возникла гипотеза, что для работы процесса в ядре LKL выделяется vma (блок памяти) размером с выделенной памятью для lk, и что этот объём не меняется при работе приложения, то есть, в процессе работы LKL к

внешнему аллокатору памяти не обращается. Для подтверждения этого предположения было проведено исследование поведения программы lklfuse.

2.2.3 Исследование с помощью отладчика

Исследование с помощью отладчика заключается в запуске программы, использующей LKL API, например программы libfuse (tools/lkl/lklfuse.c) и в анализе с помощью gdb какие функции для работы с памятью вызываются.

Исследование с помощью отладчика показало, что при вызове __get_free_pages() не вызывается никаких специфичных для LKL функций (это проверить достаточно легко, в силу того, что все обращения к “внешнему миру” в lkl организованы как вызовы элементы таблицы функций lkl_host_ops. Таким образом, можно сделать вывод, что механизм получения страниц такой же, как и обычном ядре Linux. Специфичный для LKL код написан в arch/lkl(код, который компилируется в подсистеме сборки ядра) и tools/lkl/lib() и там никакой специфики, связанной с получением страниц, нет.

В ходе отладки было замечено только то, что вызываются обёртки к функции malloc только во внешних, по отношению к ядру функциях (callbacks для ядра хостовой ОС), причем выделенные хостовой ОС ресурсы используются для внутренних потребностей LKL.

2.3 Поддержка потоков

LKL не предоставляет услуг по созданию потоков для программ его использующих. Однако ядру Linux(внутри LKL) требуются потоки для корректной и эффективной работы. В ранних версиях LKL была сделана эмуляция внутренних потоков внутри LKL.

Но главная проблема с с внутренними потоками заключается в том, что у них должен быть раздельный стек, а выделение стека осуществляет хостовая операционная система. В результате LKL не может выделить новый стек для потока, и приходится дробить на кусочки выделенный хостовой операционной

системой стек. В результате его размер для каждого потока уменьшается пропорционально числу потоков в ядре LKL, что накладывает жесткие ограничения на число эмулируемых потоков.

Поэтому разработчики решили отказаться от идеи эмулировать потоки внутри LKL. Теперь если ядру Linux нужны потоки, то LKL просит у хостовой ОС создать поток, реализовав 2 промежуточных метода, один из которых вызывает системный вызов thread_create, а другой вызывает системный вызов thread_exit.

Полное описание предоставляемых архитектуре LKL ядра Linux можно найти в файле:

linux/tools/lkl/include/uapi/host-ops.h

```
	lkl_thread_t (*thread_create)(void (*f)(void *), void *arg);
	void (*thread_detach)(void);
	void (*thread_exit)(void);
	int (*thread_join)(lkl_thread_t tid);
```

В данном файле объявлены функции обратного вызова(callback), которые вызывает ядро Linux внутри LKL. Реализацию всех функций обратного вызова можно найти в документа linux/tools/lkl/lib/posix-host.c (posix-совместимые системы) или linux/tools/lkl/lib/nt-host.c (Windows).

Вот например как выглядит функция обратного вызова для создания нити.

```
static lkl_thread_t thread_create(void (*fn)(void *), void *arg)
{
	pthread_t thread;
	if (WARN_PTHREAD(pthread_create(&thread, NULL, (void* (*)(void *))fn,
arg)))
		return 0;
	else
		return (lkl_thread_t) thread;
}
```

Эти функции обратного вызова будут исполнены ядром хостовой ОС, выделенные ресурсы будут предоставлены ядру Linux внутри LKL.

Однако этот подход, решая проблему со стеком порождает новую проблему: ядро Linux должно быть уверено, что именно оно управляет переключением потоков, а при данном подходе переключением потоков управляет хостовая операционная система. Чтобы это решить эту проблему, в LKL был придуман механизм эмуляции управления потоками.

2.4 Управление потоками

Как можно в LKL создать эмуляцию управления потоками? Накладывается специальное ограничение: в каждый момент времени LKL может работать только в одном потоке. Данное ограничение существенно снижает производительность ядра, но сильно упрощает реализацию системы потоков. Однопоточность достигается указанием соответствующей инструкции в конфигурационном файле. Мы явно говорим что ядро собирается для однопроцессорной машины (опция CONFIG_SMP). LKL получает поток из внешней системы, и получает соответствующий ему системный семафор. Поскольку данный ресурс получается из хостовой ОС, то реализовано это получение точно также через функцию обратного вызова.

Планировщик Linux внутри LKL переписан так что у него есть два слоя: один слой системный(для этого мы получаем от системы семафоры), второй слой - внутренний, принадлежащий LKL (который заведует логикой управления потоков внутри LKL).

1. В архитектурно-зависимой части ядра общие для ядра Linux конструкции (блокировки, семафоры, потоки) реализуются с использованием внешних функций обратного вызова, предоставляющиеся через lkl_host_ops.
2. Для внутренних целей архитектурно-зависимой части ядра функции обратного вызова из lkl_host_ops вызываются напрямую.

Чтобы разобраться с тем как и где добавлен слой отвечающий за логику управления потоками, а также за получение внешних ресурсов(потоков и семафоров) от хостовой ОС проанализируем главную функцию планировщика __schedule(), реализацию которой находится в файле kernel/shed/core.c

Данная функция является ключевой функцией планировщика. Вход в данную функцию осуществляется в следующих случаях:

1. Явная блокировка: взаимные исключения, семафоры, ожидания в очереди.
2. Прерывания и возврат из пользовательских процессов.
3. Пробуждение потока не приводит к вызову __schedule().

При переключении процессов планировщик помимо прочей работы занимается переключением контекстов. В функции __shedule() вызывается функция context_switch(rq, prev, next), которая в свою очередь вызывает макрос switch_to(prev, next, last). определённый в документе linux/include/asm-generic/switch_to.h :
#define switch_to(prev, next, last) в ходе работы которого вызывается функция __switch_to(struct task_struct *, struct task_struct *), которая уже находится в специфичном именно для LKL файле(linux/arch/lkl/kernel/threads.c). Именно здесь и реализуется логика управления потоками в ядре Linux внутри LKL.
При создании потока ядром Linux внутри LKL вызывается функция обратного вызова таким образом: lkl_ops->sem_alloc(0) и заносится в структуру threadinfo, которая определена в
arch/lkl/include/asm/thread_info.h
Эта структура хранит в себе информацию о потоке. Одно из полей структуры void *sched_sem - тот самый семафор, который внутренний планировщик потоков будет использовать для старта/остановки потока. При инициализации каждому новому потоку ставится в соответствие семафор, выделенный хостовой операционной системой, со значением равным нулю:
ti->sched_sem = lkl_ops->sem_alloc(0)
Таким образом только что созданный семафор блокируется до тех пор, пока планировщик LKL не примет решение о запуске данного потока.

Основная работа по модификации планировщика реализована в виде функции struct task_struct *__switch_to(struct task_struct *prev, struct task_struct *next). Реализацию этой функции можно найти в linux/arch/lkl/kernel/thread.c
Основные три этапа работы функции:

1)сохранить указатель на task_struct того потока с которого происходит переключение

2)разблокировать с помощью вызова соответствующей функции обратного вызова тот поток, на который планируется выполнить переключение:
lkl_ops->sem_up(_next->sched_sem);

3)блокировать текущий поток из которого планируется выполнить переключение:
lkl_ops->sem_down(ei.sched_sem);

Функция __switch_to возвращает указатель на task_struct того процесса из которого была вызвана функция __switch_to. Причём шаг 1 в котором сохраняется указатель на task_struct того потока с которого происходит переключение обеспечивает то, что возвращаться всегда будет указатель на task_struct переключаемого процесса.

Пример: пусть в данный момент времени ядро Linux внутри LKL работает в потоке №1, вызывается функция __switch_to внутри которого происходит переключение на поток №2 с помощью sem_up, далее поток №1 блокируется вызывая sem_down, причём перед этим поток №1 сохраняет указатель на свой task_struct в общую для всех вызовов функции __switch_to переменную abs_prev. В результате функция __switch_to вернёт указатель на task_struct потока №1. Пусть в ходе дальнейшей работы происходит переключение на поток №3. И пусть поток №3 в своём вызове функции __switch_to будит поток №1. Таким образом функция вернёт указатель на task_struct потока №3, при этом в данный момент времени будет исполняться поток номер №1.

Таким образом, когда внутреннему планировщику потоков нужно переключиться на новый поток, он закрывает внешний семафор для текущего и открывает внешний семафор для нужного ему потока. Так как при этом все остальные потоки закрыты семафором, планировщик хостовой системы может запустить только один поток, тот самый который нужен планировщик потоков внутри LKL. Если этого не делать, внешний планировщик потоков может запускать потоки в порядке, который ему подсказывает его логика, и планировщик потоков внутри LKL теряет контроль над порядком запуска потоков, что ломает внутреннюю логику работы ядра Linux. При этом внутренний слой LKL работает как обычно, блокирует синхронизируемые потоки и т.д. Планировщик потоков внутри LKL открывает и закрывает внешние блокировки в соответствии с теми запросами на блокировку/запуск потока, которые получает от своего ядра.

2.5 Системные вызовы

2.5.1 Общая информация о системных вызовах

Системным вызовом называется обращение прикладной программы к ядру операционной системы для выполнения какой-либо операции.
Можно сказать, что это прослойка, которая вызывает некоторую функцию из пространства ядра(kernel space), разрешённую для вызова из пространства пользователя(user space). Но в данном случае необходимо сделать важное уточнение.
Для начала следует уточнить что такое user space и kernel space. Рассмотрим виртуальную память процесса. Диапазон адресов виртуальной памяти начинается от 0 до какого-то максимального значения, определяемого разрядностью ОС. На 32битной системе предел – это 4Гб. В случае с линуксовым ядром само ядро в памяти располагается начиная от какого то определённого адреса вверх.

По умолчанию, ядро начинается от метки 3ГБ. В LKL так как мы не контролируем страничную адресацию, ядро начинается с некого смещения в блоке памяти, который выделен для его работы.

На рис.2 изображена виртуальная память которая существует в рамках одного процесса.

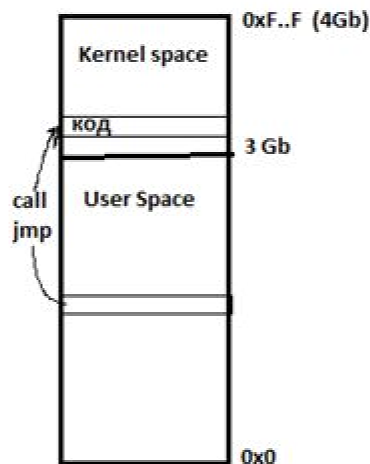

рис.2

Задача системного вызова - выполнить передачу управления от одного участка кода к другому участку кода. Как в архитектуре x86 происходит передача управления от одного участка кода к другому?

Такими способами являются: системный вызов, прерывание, безусловный переход, вызов подпрограммы (инструкции sysenter, int, jmp, call).

В таком случае возникает закономерный вопрос, чем именно системные вызовы отличаются от обычных вызовов функции.

Чтобы ответить на этот вопрос изобразим виртуальную память от адреса A до адреса B на рис.3

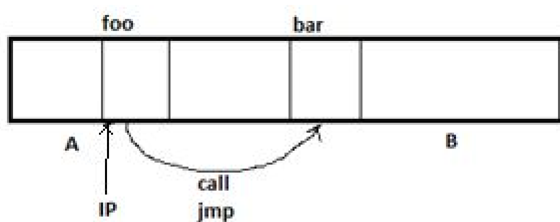

Рис.3

Для ответа ответа на поставленный вопрос следует уточнить что на ассемблерном языке означает утверждение о том, что выполняется функция foo.
У процессора есть регистр IP(Instruction Pointer). Процессор указывает им в точку кода, который относится к функции foo и выполняет инструкции, соответствующие этой функции. Процессор читает инструкции из этого куска памяти, который относится к функции foo на рис.3, и выполняет эти инструкции, одновременно изменяя регистр IP.

Причём для того, чтобы передать управление на функцию bar нужно записать в стек выполнения адрес куда нужно вернуться(так называемый адрес возврата). Для этого есть инструкция call.

Чем jump отличается от call? jump не записывает адрес возврата в стек.
Итак, обычный вызов: применяем инструкцию call - переключается IP, процессор выполняет подпрограмму, и по оператору ret возвращается к следующей инструкции после оператора вызова.

Теперь можно ответить на вопрос: "В чём заключается существенное отличие системного вызова от обычного вызова функции?" Системный вызов подразумевает прерывание контекста текущего процесса с целью передачи управления ядру операционной системы. При этом нужно позаботиться о том, чтобы не осуществилась передача приложению прав выполнения ядра. Отличие системного вызова от обычного вызова функции заключается в переключении привилегий. Именно по этой причине с точки зрения ассемблерных инструкций системный вызов осуществляется не через jmp или call.
С точки зрения ассемблерных инструкций системный вызов - это вызов программного прерывания. Соответствующая инструкция называется int.
Также есть ещё инструкция sysenter.

2.5.2 Системные вызовы в LKL

Для изучения деталей реализации системных вызовов в LKL описаны в документе: linux/arch/lkl/kernel/syscalls.c

Есть таблица системных вызовов, которая заполняется с помощью данного макроса:

```
syscall_handler_t syscall_table[__N+R_syscalls] = {
	[0 ... __NR_syscalls - 1] =  (syscall_handler_t)sys_ni_syscall,
```

Структура используемая для описания системного вызова выглядит следующим образом:

```
struct syscall {
	long no, *params, ret;
};
```

long no - номер системного вызова

long *params - указатель на параметры, которые передаются системному вызову

long ret - возвращаемое значение

Для большинства компиляторов, используемых в posix-системах sizeof(long)==sizeof(void *). К сожалению, на этот факт рассчитывают и авторы ядра Linux, хотя стандарт языка C ничего не говорит о размере данного типа, кроме выполнения следующего неравенства: sizeof(int)<=sizeof(long)<=sizeof(long long). В качестве примера системы, в которой sizeof(long) не равен sizeof(void *) выступает 64битная сборка Windows. Именно в этом заключается проблема при сборке LKL для 64битной версии Windows. Проблема возникает не в самой подсистеме LKL, а в текстах ядра Linux, предполагающих указанное выше равенство.

Для выполнения системных вызовов используется отдельный поток и структура syscall_thread_data. В эту структуру входит информация о системном вызове, блокировки, информация об очереди вызовов и служебные флаги.

Разберёмся с тем, как происходит обработка системных вызовов.

Существует очередь(queque) системных вызовов. Из этой очереди выбирается очередной системный вызов для обработки с помощью функции dequeue_syscall. Эта функция атомарно забирает значение. Для атомарного забора значения используется примитив __sync_fetch_and_and, благодаря которому есть возможность повесить только один обработчик на рассматриваемый системный вызов.

Для вызова нужного системного вызова из таблицы системных вызовов используется функция run_syscall.
В коде данной функции ключевым моментом является вызов нужного системного вызова из таблицы и запуск функции task_work_run(), которая обрабатывает все задачи, которые лежат в очереди. Под задаей понимается структура, описывающая процесс в ядре Linux.
В ядре есть специальный объект, он называется task struct – эта структура описывающая процесс. И хотя вот это называется struct - это называют объектом, он описывает некий процесс. Рассмотрим основные поля этого объекта, которые нужны чтобы описать процесс с точки зрения ядра.

pid, - идентификатор процесса
uid, gid, - идентификаторы пользователя и его группы
ftable – здесь лежит ссылка на объект, который называется ftable (таблица открытых файлов)
signal, sighand, - обработчики сигналов
parent - это не просто pid родителя, это именно ссылка на объект-родитель
mm – специальный объект, который описывает виртуальное адресное пространство этого процесса
В этой структуре есть и другие поля, выше выделены основные.

Возвращаясь к описанию работы функции task_work_run() следует уточнить, что подобная обработка очереди с использованием примитивов синхронизации таких как инструкция cmpxchg и барьер памяти является типичной для Linux (не только для LKL).

Таким образом на данном этапе рассмотрен механизм обработки системных вызовов в LKL.

2.5.3 Использование системных вызовов в LKL

Так как заголовки для кода уровня ядра и уровня пользователя с друг другом несовместимы и перемешивать их нельзя ни в коем случае, то в LKL используют следующий подход.

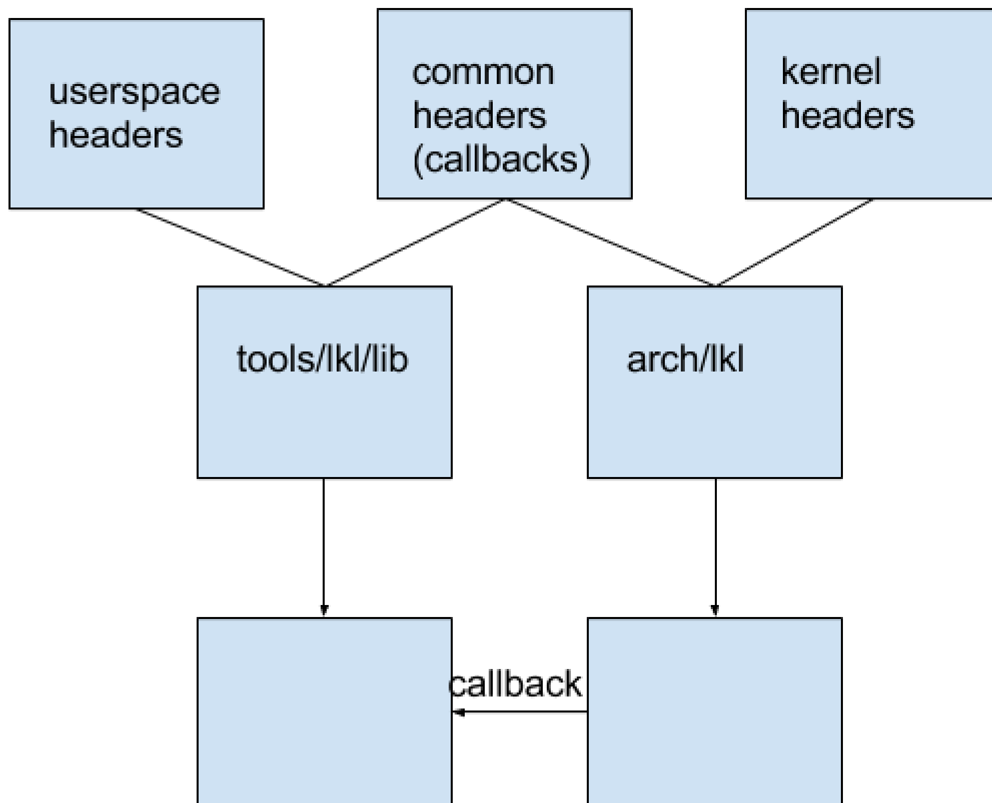

рис.4

Как уже было сказано выше, библиотека LKL разделена на 2 существенно разных блока: ядро LKL и инструменты по осуществлению связи ядра с приложениями и

окружением. Известно, что имеется конфликт между включаемыми файлами сборки ядра и включаемыми файлами пользовательского уровня. Блочная архитектура библиотеки LKL сводит эти проблемы до минимальных, так как единственными общими включаемыми файлами должны быть файлы, описывающие протокол взаимодействия между архитектурно-зависимой частью ядра и внешним окружением (реализация “псевдосистемных вызовов” и функций обратного вызова, и структур, с ними связанных).
Такая модульность также позволяет использовать LKL для систем с архитектурой, достаточно сильно отличающейся от Unix-систем.

Пример использования данного потока разделения заголовочных фалов из пространства ядра и пространства пользователя можно пронаблюдать в следующем документе:
linux/tools/lkl/lib/posix-host.c можно встретить функцию
static lkl_thread_t thread_create(void (*fn)(void *), void *arg), адрес которой заносится в поле thread_create структуры, на которую указывает lkl_host_ops.

Поскольку LKL запускается в непривилегированном процессе пользователя, то аппаратные возможности выделения ресурсов ядру LKL не доступны. Таким образом, внутри LKL планировщик зависит от внешней ОС, которая опосредованно и управляет потоками. Для этих целей в структуре lkl_host_ops реализован интерфейс работы с потоками, функции которого реализованы в модулях posix-host(или nt-host) Данный код подменяет то, что можно сделать аппаратным образом вызовом соответствующих системных функций.

Как уже было упомянуто выше, для выполнения запросов к внешнему миру (те операции, которые не предоставляет оборудование, и которые должны быть реализованы программно) архитектурно-зависимая часть ядра осуществляет с помощью вызовов из таблицы lkl_host_ops. (Другая часть функий обратного

вызова (callback) используются в драйвере виртуального устройства. Рассмотрим их позднее в разделе 3. Реализация).

```
struct lkl_host_operations lkl_host_ops = {
        .panic = panic,
        .thread_create = thread_create,
        .thread_exit = thread_exit,
        .sem_alloc = sem_alloc,
        .sem_free = sem_free,
        .sem_up = sem_up,
        … };
```

Данную таблицу использует подсистема arch/lkl ядра.

API для пользовательских приложений состоит из набора “псевдосистемных” вызовов и служебных функций, осуществляющих следующую функциональность:

1. Запуск/остановка ядра LKL
2. Создание виртуальных устройств (virtio)

Примером работы приложения с библиотекой LKL является приложение test_tool.c (приложение использующее LKL API и работающее с файловыми системами поддерживаемыми в Linux, которое было написано в практической части дипломной работы(см раздел 3. Результаты)).

Если обратиться к реализации данных заголовочных файлов в библиотеке LKL, можно заметить, что они не включают никаких заголовочных файлов из пространства ядра.
В ходе исследования интерфейса системных вызовов было выяснено что несмотря на то, что приложение, использующее LKL, технически способно делать прямые вызовы к любой функции экспортируемой ядром(в силу отсутствия разделения памяти и уровней привилегий), это противоречит архитектуре ядра и может привести к непредсказуемым последствиям в силу того, что будут

пропущены механизмы защиты ядра. Также такой подход легко может к нарушению логики работы ядра.
API который LKL предоставляет приложению подмножество системных вызовов Linux. Приложение должно только использовать только публичный kernel API: системные вызовы и служебные функции библиотеки LKL. Так как у LKL нет поддержки симметричного мультипроцессирования(SMP), то у приложения не будет возможности напрямую вызывать обработчики системных вызовов благодаря возможным состязательным ситуациям между обработчиком системного вызова, ядерными потоками, обработчиками прерываний, или другими параллельными обработчиками системных вызовов.

Обработчик прерываний/системных вызовов добавляет все запросы на системный вызов в очередь обработки (work_queue). Обычно, когда ядро Linux заканчивает свою инициализацию, оно запускает процесс init, но LKL не может это сделать так как LKL не поддерживает процессы в пространстве пользователя. Вместо этого LKL запускает специально предназначенную процедуру, которая дожидается событий из очереди системных вызовов. На каждый запрос данная нить вызывает соответствующий обработчик системного вызова, кладёт значение, возвращаемое обработчиком в определённое поле структуры, связанной с системным вызовом и переключается в вызывающую нить. Будучи разблокированной, обёртка для системного вызова в вызывающей нити освобождает соответствующую структуру и возвращает результат тому, кто вызвал.

3. Реализация

3.1 Формулировка задачи в терминах LKL

Для начала необходимо понять каким образом задача по исследованию шаблонов нагрузок на файловые системы может быть реализована при помощи программного комплекса Linux Kernel Library.
Изначальная постановка задачи заключалась в исследовании шаблонов нагрузок на файловые системы. Разные программы по-разному используют файловые системы. Хочется иметь возможность собирать статистическую информацию об этом использовании. Собранная статистика должна быть такой, чтобы по ней можно было пытаться "воспроизводить" нагрузку.
Цель сбора этой статистики - возможность модификации аллокатора блоков. Части драйвера файловых систем, которая выполняет запросы к диску. В файловых системах самое проблемное место это аллокатор блоков. Если он работает плохо, то при работе с файловой системой у нас получается много произвольных запросов к диску. Если он работает хорошо, то запросы к диску в основном последовательные. Статистика должна нам давать достаточно информации, чтобы понимать хорошо или плохо работает аллокатор блоков.

Таким образом задача по изучению шаблонов нагрузок на файловые системы в терминах LKL сводится к разработке инструментария, который может собирать статистику, воспроизводить её и помогать ответить на вопрос хороший у нас аллокатор или нет.

Какими именно критериями пользоваться при оценке качества работы аллокатора блоков?
Это тоже тема для исследований. "Хорошим" мы будем считать такой аллокатор, при котором результирующая нагрузка на диск будет максимально последовательной. Однако, всегда можно таким образом обращаться к файлам,

что в независимости от их размещения на диске, нагрузка будет на диск случайной. Таким образом, нас интересует сценарий типовой нагрузки на файловую систему.
Цель данного исследования в первую очередь это создания тестового стенда который который будет осуществлять сбор статистики работы с файловой системой, чтобы по этой статистике можно было понять какие именно секторы диска участвуют в данных операциях.

То есть в ходе исследование изучается статистика обращения к диску, так как именно она показывает, насколько последовательно идёт обращение к секторам диска.

3.2 Блочные устройства в LKL

В LKL создаются так называемые виртуальные блочные устройства, которые предоставляют функции обратного вызова посредством которых происходит работа с реальными файлами или блочными устройствами, которые находятся внутри хостовой ОС.
Доступ к виртуальным блочным устройствам в LKL осуществляется с помощью передачи им следующей структуры, описывающей набор операций, которые будут произведены на блочном устройстве:

```
struct lkl_dev_buf {
        void *addr; /* адрес, по которому будет произведены чтение/запись */
        unsigned int len; /* размер блока данных */
};
struct lkl_blk_req
{
     unsigned int type; /* тип операции: чтение/запись, синхронизация */
     unsigned int prio; /* приоритет */
     unsigned long long sector; /* номер стартового сектора */
     struct lkl_dev_buf *buf; /*массив операций */
     int count; /* число операций*/
};
```

Функция blk_request - это функция, обрабатывающая запросы к блочному устройству указанного выше вида:
static int blk_request(union lkl_disk disk, struct lkl_blk_req *req)

Образы диска подключаются к LKL через псевдоустройства virtio следующим образом:

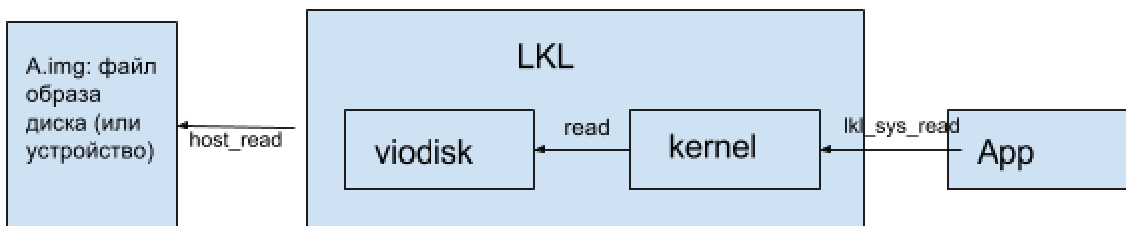

рис.5.

На рис.5 показано первое приближение механизма взаимодействия приложения, LKL, виртуального блочного устройства, которое сопоставляется реальному блочному устройству и файла, лежащего на реальном блочном устройстве.
Мы ещё вернёмся описанию этого взаимодействия с целью его уточнения в пункте 3.2 Интегрирование логирующей функции в LKL

3.3 Исследование возможности работы с файловыми системами в LKL.

В рамках работы с этим пунктом было проведено исследование возможности использования LKL для работы с файловыми системами, поддерживаемыми ядром Linux.

Был проанализирован интерфейс, предоставляемый LKL, на основе которого была написана утилита для работы с директориями и файлами в разных файловых системах (поддерживаемых Linux), например, в ext4 и vfat.

Утилита может прочитать содержимое директорий по имени:

test_tool [имя файла или устройства] [имя файловой системы] [каталог]
Она создаёт виртуальное блочное устройство, которое связано с реальным устройством (или каким-то образом файловой системы).
LKL, обращаясь к этому виртуальному устройство, тем самым обращается к реальному объекту хостовой ОС (к файлу или устройству) драйвером файловой системы, находит директорию по имени, читает её содержимое и выводит содержимое на экран.
Схема взаимодействия показана на рис.5 в предудщем пункте 3.1, посвящённому исследованию функционирования блочных устройств в LKL.

3.4 Интегрирование логирующей функции в LKL

Будем собирать статистику не обращая внимания на то, что логирование каждого обращения в диску - накладная операция, так как мы измеряем не время операций, а последовательность обращения к диску.
Функция логирующая каждое обращение к секторам диска принимает на вход адрес структуры lkl_blk_req описанной в предыдущем пункте.
Функция протоколирования принимет на вход структуру vlad_log_rec,

```
typedef struct
{
        unsigned int type;
        unsigned int prio;
        unsigned long long sector;
        int count;
} vlad_log_rec;
```

В ней суммируется обращение из всех буферов структуры lkl_blk_req.
Для осуществления протоколирования используется функция
void vlad_log(const vlad_log_rec * req);
Для корректного функционирования системы протоколирования до старта ядра lkl необходимо вызвать функцию:
int vlad_init_log(const char * fname,size_t size);

Ее параметры: fname - имя файла протокола, size - начальный размер журнала. Этот параметр может быть задан в произвольное разумное значение, так как:

1. Если размера журнала будет недостаточно, он будет увеличен.
2. После окончания работы журнал обрезается по последней записи.

Окончание протоколирования осуществляется функцией

```
void vlad_close_log(void);
```

Модификации которые необходимо внести в программу использующую LKL API для работы с файловой системой (например, linux/tools/lkl/lklfuse.c) :

```
static int start_lkl(void)
{
        long ret;
        char mpoint[32];
+       vlad_init_log("/home/vlad/vlad.log",1024*1024);
        ret = lkl_start_kernel(&lkl_host_ops, lklfuse.mb * 1024 * 1024, "");
...
static void stop_lkl(void)
{       int ret;
+       vlad_close_log();
…
```

Опишем подробнее функцию vlad_log:

У этой функции есть interlocked счётчик в какое место записывать, кусок памяти, представленный как массив, ассоциированный с файлом на диске (используя системный вызов mmap). Массив состоит из элементов типа vlad_log_rec. В этот массив мы будем записываем последовательно структуры vlad_log_rec.

Как было сказано выше, этот массив перед запуском LKL маппируется на какой-то файл. И в итоге, по выходу из этого LKL в файле находятся подряд записанные номера блоков, над которыми выполняются определённые действия. Обработчик блочных запросов на псевдоустройство модифицирован следующим образом: мы аккумулируем данные об операции, записываем их в протокол и вызываем старый блочный обработчик.

Напомним, что образ диска подключаются к LKL через псевдоустройство virtio образом, показанным на рис.5(см. раздел 3.1 Блочные устройства в LKL)

Для организации такого псевдоустройства есть API, которое по файловому дескриптору/HANDLE создает заготовку для устройства, которая, в свою очередь преобразуется в устройство. Соответственно, в метод доступа к этому виртуальному устройству и был встроен механизм протоколирования, который анализирует запросы (чтение, запись,...) и заносит информацию в журнал о секторах, с которыми будет проводиться данная операция

3)Теперь можно запустить lklfuse (драйвер файловой системы уровня пользователя, транслирующий вызовы fuse в вызовы LKL API), нагрузить подмонтированную таким образом файловую систему => появляется файл, в котором есть информация об обращении к диску. Ее анализом можно получить, насколько последовательным было обращение к секторам диска. Таким образом, можно реализовывать произвольные сценарии работы с файловой системой и получать данные о качестве аллокации блоков.

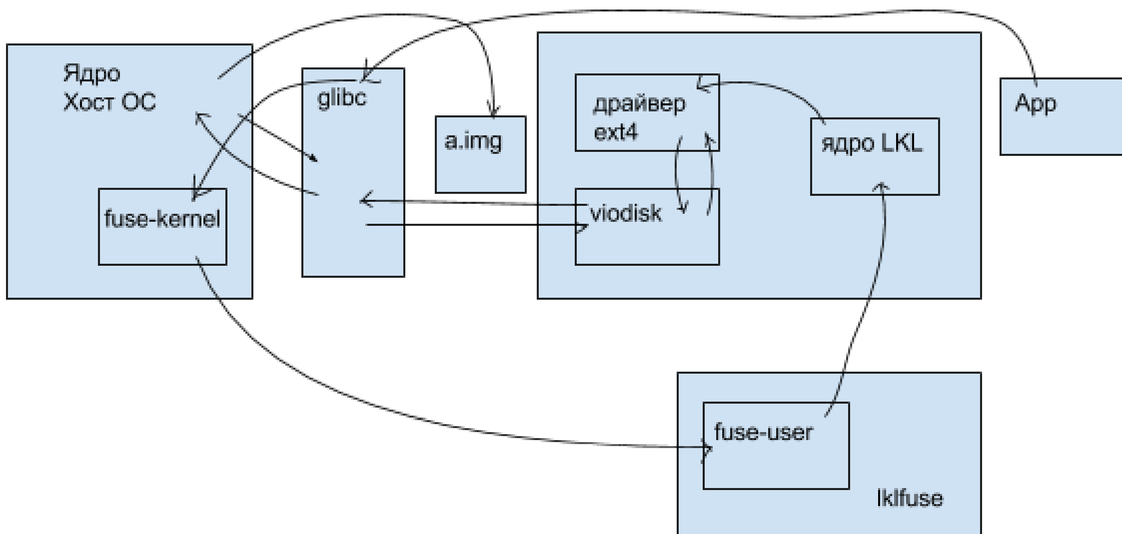

Рис.6

На рис.6 более подробное описание механизма взаимодействия приложения, LKL, виртуального блочного устройства, которое сопоставляется реальному блочному

устройству и файла, лежащего на реальном блочном устройстве. Функция, vlad_blk_rec логирует обращение от драйвера файловой системы(например ext4) к виртуальному блочному устройству, которое в свою очередь вызовет функцию обратного вызова на соответствующую операцию, который будет выполнен ядром хостовой ОС. Информация о том, какие операции драйвер файловой системы запросил у блочного устройства записывается в журнал, который будет подвержен анализу в следующем пункте.

В более ранних версиях сбора статистика использовалась при помощи приложения использующего LKL API(типа test_tool), но впоследствии было принято решение драйвер lklfuse, который расширяет универсальность тестов нагрузки на файловую систему.

3.5 Нагрузка файловой системы

На этом этапе выполняется запуск эмулятора на какой-то файловой системе и нагрузка этой файловой системы с помощью приложения использующего LKL API.

Исследуемые файловые системы были подвержены следующим тестам.

тест №1:

Копирование всех файлов из домашней директории в специальное устройство dev/null

тест№2:

Копирование директора с проектом LKL на тестируемое блочное устройство и выполнение команды make

тест№3:

Программа, которая в случайном порядке создаёт директории, файлы, дописывает информацию в файлы, удаляет эти файлы.

3.6 Сбор статистики и анализ получившейся информации

В предыдущих пунктах было приведено описание процесса интеграции логирующей функции в LKL. Также был описан механизм работы эмулятора при различных типах нагрузки блочного устройства. На данном этапе имеется журнал, данные в котором следует подвергнуть обработке и анализу с целью извлечения информации позволяющей ответить на вопрос о частоте последовательных обращений в блочному устройству.

После проведения необходимого анализа и обработки были получены данные следующего вида:

-16	118
-8	6
-6	1
-4	1
0	1070808
8	3362
16	1548
24	703

столбец 1 показывает количество секторов не которой пришлось сместиться при очередном обращении к блочному устройству(к диску)

столбец 2 показывает сколько раз пришлось сместится на данное число секторов в процессе нагрузки блочного устройства(диска)

Итак, был получен набор чисел следующего вида:

0, 0, 0, -6, 100, 1000, 0, 0, и т.д.

“0”: что обращение к диску было последовательным,

“-6”: произошёл скачок на 6 секторов назад,

“1000”: произошёл скачок на 1000 секторов вперёд.

Отсюда можно получить количество последовательных обращений в процентном отношении ко всем обращениям.

3.7 Полученная статистика

Исследуемые файловые системы были подвержены следующим тестам описание которых можно найти в предыдущем пункте(3.6 Сбор статистики и анализ получившейся информации)

ТЕСТ №1

№1	**$< -10^6$**	**$(-10^6; -10^5)$**	**$(-10^5; -10^4)$**	**$(-10^4; 0)$**	**0**	**$(0;10^4)$**	**$(10^4; 10^5)$**	**$(10^5; 10^6)$**	**$>=10^6$**
btrfs	0,0013	0,0034	0,0009	0,0020	0,9827	0,0040	0,0010	0,0034	0,0013
ext2	0,0043	0,0016	0,0009	0,0028	0,9713	0,0099	0,0017	0,0029	0,0046
ext3	0,0014	0,0030	0,0011	0,0118	0,9697	0,0066	0,0016	0,0033	0,0014
ext4	0,0017	0,0012	0,0009	0,0058	0,9722	0,0137	0,0012	0,0014	0,0019
xfs	0,0044	0,0006	0,0008	0,0027	0,9750	0,0084	0,0014	0,0009	0,0058

ТЕСТ №2

№2	**$< -10^6$**	**$(-10^6; -10^5)$**	**$(-10^5; -10^4)$**	**$(-10^4; 0)$**	**0**	**$(0;10^4)$**	**$(10^4; 10^5)$**	**$(10^5; 10^6)$**	**$>=10^6$**
btrfs	0,0000	0,0024	0,0008	0,0032	0,9881	0,0008	0,0024	0,0024	0,0000
ext2	0,0099	0,0006	0,0000	0,0112	0,9227	0,0266	0,0000	0,0199	0,0090
ext3	0,0054	0,0030	0,0000	0,2656	0,7128	0,0036	0,0000	0,0039	0,0057
ext4	0,0005	0,0005	0,0005	0,0033	0,9877	0,0057	0,0005	0,0005	0,0009
xfs	0,0110	0,0000	0,0004	0,0123	0,9484	0,0066	0,0004	0,0000	0,0207

ТЕСТ №3

№3	**$< -10^6$**	**$(-10^6; -10^5)$**	**$(-10^5; -10^4)$**	**$(-10^4; 0)$**	**0**	**$(0;10^4)$**	**$(10^4; 10^5)$**	**$(10^5; 10^6)$**	**$>=10^6$**
btrfs	0,0001	0,0004	0,0001	0,0001	0,9856	0,0129	0,0001	0,0004	0,0001
ext2	0,0014	0,0003	0,0003	0,0016	0,9744	0,0186	0,0011	0,0007	0,0017
ext3	0,0005	0,0002	0,0002	0,0019	0,9787	0,0171	0,0006	0,0004	0,0005
ext4	0,0002	0,0002	0,0001	0,0015	0,9815	0,0159	0,0002	0,0002	0,0002
xfs	0,0002	0,0001	0,0001	0,0015	0,9816	0,0160	0,0002	0,0001	0,0002

4. Заключение

4.1 Результаты

Был проанализирован интерфейс, предоставляемый LKL, на основе которого была написана утилита для работы с директориями и файлами в разных файловых системах (поддерживаемых Linux), например, в ext4.

Интегрирована в LKL возможность сбора статистики использования файловой системы на уровне обращения к диску.

4.2 Перспективы развития

В ходе анализа работы аллокатора блоков в различных файловых системах было получен вывод о том, что запросы при обращения драйверов для все рассмотренных файловых систем к диску в основном последовательные. Имеет смысл усложнить тесты чтобы понимать как оптимизировать аллокатор блоков, тестировать эти оптимизации или даже попытаться сделать адаптивный (под нагрузку) аллокатор.

4.3 Возможное применение

Результатом работы является тестовый стенд, позволяющий исследовать поведение произвольной файловой системы под различными типами нагрузки без необходимости загружать ядро на физическом сервере или в виртуальной машине.

Список литературы

1. Purdila O., Grijincu L., Tapus N. “LKL: The Linux Kernel Library,” Available from https://www.researchgate.net/publication/224164682_LKL_The_Linux_kernel_library, 2010.
2. Source code of LKL, Available from https://github.com/lkl
3. Love R. “Linux Kernel Development (Second Edition),” – 2005.
4. Paul R. “Linux kernel in 2011: 15 million total lines of code and Microsoft is a top contributor,” Available from http://arstechnica.com/business/2012/04/linux-kernel-in-2011-15-million-total-lines-of-code-and-microsoft-is-a-top-contributor/, 2012.