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 _____ Ондар Адыгжы Менгиевич

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

Москва 2016

Оглавление

1. Введение

1.1. Linux Kernel Library (LKL)

Ядро ОС 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 (*virtual file system*), управление задачами (*task management*) или планирование выполнения процессов (*scheduling*) [1]. Однако в рамках данной работы рассматривается возможность использования драйверов файловых систем, поддерживаемых ядром ОС Linux. Более того, LKL может использоваться в разных окружениях (Linux, Windows), если данная среда предоставит некоторые операции для данной библиотеки (например, функции выделения и освобождения памяти).

1.2. Цели LKL

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

- LKL не должен требовать определенного конкретного окружения (*environment*), на которой будет использоваться эта библиотека;
- 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 окружения, 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 операций[1], характерных для данного окружении, где LKL будет запускаться.

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

1.3. Общая схема LKL

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

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

Иллюстрация общей картины LKL [1]:

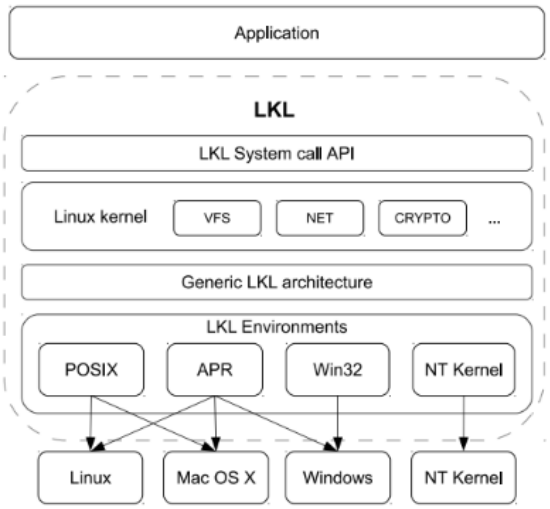

Рис. 1. Общая схема LKL

[1] Эти операции называются *native operations* [1], более подробно см. раздел «2.3. Операции LKL, предоставляемые окружением»

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

2. Архитектура LKL

В рамках данной работы в этом пункте рассмотрим некоторые важные компоненты LKL[3].

2.1. Подсистема IRQ

В LKL реализована подсистема IRQ. Всего разных номеров прерываний LKL может выделить:

```
#define NR_IRQS ((int)sizeof(long) * 8)
```

Чтобы определять, какое именно прерывание произошло, LKL использует переменную, типа *unsigned long* (собственно, поэтому в *NR_IRQS* фигурирует *sizeof(long)*).

```
static unsigned long irq_status;
```

Каждый бит этой переменной соответствует некоторому номеру прерывания (0-вой бит – прерыванию №1, 1-ый – прерыванию №2, и т.д.): если бит установлен, значит, произошло соответствующее прерывание, и необходимо его обработать, после обработки этот бит сбрасывается. Определены макросы для работы с этой переменной:

```
#define TEST_AND_CLEAR_IRQ_STATUS(x)    __sync_fetch_and_and(&irq_status, 0)
#define IRQ_BIT(x)                      BIT(x-1)
#define SET_IRQ_STATUS(x)               __sync_fetch_and_or(&irq_status, BIT(x - 1))
```

Основная часть интерфейса IRQ системы LKL:

- lkl_trigger_irq

```
/*
* This function is used by the device host side
* to signal its Linux counterpart that some event happened.
*/
int lkl_trigger_irq(int irq);
```

[3] Исходный код LKL доступен по ссылке [2]

- lkl_get_free_irq

```
/*
* This function is called by the host device code
* to find an unused IRQ number and reserved it for its own use.
*
* @user - a string to identify the user
* @returns - and irq number that can be used by request_irq or an negative
* value in case of an error
*/
int lkl_get_free_irq(const char *user);
```

- lkl_put_irq

```
/**
* lkl_put_irq - release an IRQ number previously obtained with lkl_get_free_irq
*
* @irq - irq number to release
* @user - string identifying the user; should be the same as the one passed to
* lkl_get_free_irq when the irq number was obtained
*/
void lkl_put_irq(int irq, const char *name);
```

Рассмотрим приложение, которое, используя LKL, работает с файловой системой EXT4, например. Фактически, данное приложение будет использовать 2 драйвера устройств (*device driver*): драйвер блочного устройства Linux на стороне LKL (*Linux block device driver*) и родной (для хоста) драйвер устройства (*device driver*). Устройство на стороне LKL (*Linux device*) будет действовать как связующее звено между ядром Linux в LKL и окружением: драйвер устройства на стороне LKL (*Linux device driver*) программирует запросы ввода/вывода (*I/O requests*), вызывая родной (для хоста) драйвер устройства (для диска), который, в свою очередь, программирует аппаратные прерывания, чтобы осуществить операции ввода/вывода для физического диска.

Когда обработка такого аппаратного прерывания завершена (т.е. закончена операция ввода/вывода) на стороне хоста, драйвер устройства (диска) сообщает об этом LKL, т.е. сигнализирует об этом соответствующему драй-

веру (*Linux device driver*) для данного диска, вызывая рассмотренную функцию из интерфейса подсистемы IRQ − *lkl_trigger_irq*(). После этого обработку полученных данных (если была операция чтения, например) производит драйвер соответствующей файловой системы в ядре ОС Linux.

Описанное взаимодействие LKL, приложения, использующего данную библиотеку, и окружения проиллюстрировано на рисунке 2.

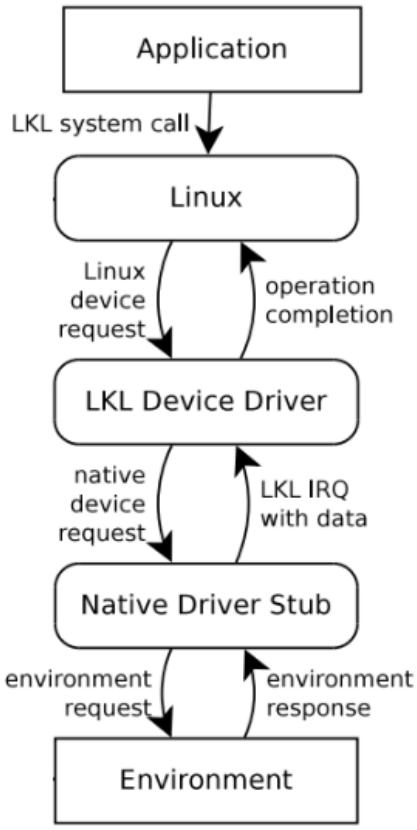

Рис. 2. Схема взаимодействия приложения, LKL и окружения

2.2. Интерфейс для диска в LKL

Диск в LKL представлен структурой lkl_disk[4].

[4] В изначальной версии lkl_disk был типа объединения (union), см. раздел «Результаты»

Для возможности работать с диском в LKL нужно его добавить (для этого есть функция *lkl_disk_add*) и смонтировать его в виртуальную файловую систему (VFS) LKL:

```
/**
* lkl_mount_dev - mount a disk
*
* This functions creates a device file for the given disk, creates a mount
* point and mounts the device over the mount point.
*
* @disk_id - the disk id identifying the disk to be mounted
* @fs_type - filesystem type
* @flags - mount flags
* @data - additional filesystem specific mount data
* @mnt_str - a string that will be filled by this function with the path where
* the filesystem has been mounted
* @mnt_str_len - size of mnt_str
* @returns - 0 on success, a negative value on error
*/
long lkl_mount_dev(unsigned int disk_id, const char *fs_type, int flags,
    void *data, char *mnt_str, unsigned int mnt_str_len);
```

Данная функция заполнит буфер *mnt_str* точкой монтирования в виртуальной файловой системе LKL.

Для размонтирования файловой системы и удаления диска (точнее для процедуры освобождения, связанной со структурой lkl_disk) существуют функции *lkl_umount_dev* и *lkl_disk_remove*, соответственно.

2.3. Операции LKL, предоставляемые окружением

Во введении было замечено, что одной из целей разработчиков LKL была возможность использования этой библиотеки на разных операционных системах. Авторы данной библиотеки выделяют некоторый набор операций (*native operations*), который нужно предоставить LKL данным конкретным окружением, чтобы запустить эту библиотеку.

Этот набор операций, зависимых от конкретной операционной системы, представлен структурой *lkl_host_operations*, поля которой являются указателями на соответствующие функции в данном окружении.

Как пример, можно рассмотреть функции создания нити в Linux и Windows:

```
// POSIX-host thread creating (see posix-host.c)
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;
}

// Windows-host thread creating (see nt-host.c)
static lkl_thread_t thread_create(void(*fn)(void *), void *arg)
{
    DWORD WINAPI(*win_fn)(LPVOID arg) = (DWORD WINAPI(*)(LPVOID))fn;

    return (lkl_thread_t)CreateThread(NULL, 0, win_fn, arg, 0, NULL);
}
```

Эти функции определены в разных файлах (*posix-host.c* и *nt-host.c*), и в зависимости от операционной системы, для которой собираем LKL, будет выбираться та или иная функция для создания нити в конкретном окружении, соответственно, поля структуры *lkl_host_operations* будут указателями на необходимые функции в этой среде.

В этом наборе операций, зависимых от окружения, также есть функции для работы с мьютексами, с таймером, для выделения и освобождения памяти (например, *malloc* и *free* для Linux).

3. Постановка задачи

Целью данной работы является исследование возможности использования LKL для работы с файловыми системами, поддерживаемыми ядром ОС Linux. Во-вторых, оценить производительность LKL при чтении и записи файлов различной длины, сравнить ее с соответствующими показателями родного окружения.

Другим интересным вопросом является анализ добавления в LKL подсистемы сбора статистики использования файловой системы. Собранная статистическая информация позволит оценить нагрузку на файловую систему.

Актуальность задачи

LKL предоставляет широкие возможности самим разработчикам ядра ОС Linux: используя данную библиотеку, можно тестировать различные компоненты ядра без необходимости загружать ядро на физической машине, в частности, драйверы разных файловых систем (в ходе исследования было замечено, что время запуска ядра в LKL и передачи управления коду драйвера EXT4 на машине с процессором AMD A10-4600M, 8Gb RAM составляет меньше секунды). Соответственно, изучение принципов работы LKL может иметь важный характер для тех, кто занимается разработкой различных подсистем, драйверов ОС Linux.

В данной же работе предпринята попытка исследовать поведение произвольной файловой системы, поддерживаемой ОС Linux, под различными типами нагрузки с использованием LKL.

Файловые системы являются областью, привлекающей особое внимание исследователей [5]. Такие вопросы в файловых системах, как размер блока, отслеживание свободных блоков, надежность и безопасность все еще далеки от идеального решения. Более того, производительность файловой

системы является одним из самых важных вопросов в современных компьютерных системах, учитывая то, насколько сильно различаются скорости операций чтения из памяти и из диска.

Соответственно, имея возможность собирать некоторую статистическую информацию о нагрузке на файловую систему, во-первых, разработчики могли бы оптимизировать ее работу. Во-вторых, используя эту статистику, можно пытаться «воспроизводить» нагрузку, другими словами, появляется возможность тестировать эти оптимизации. Более того, с использованием LKL такое тестирование возможно без необходимости загружать ядро на физической или виртуальной машине.

4. Результаты

В ходе исследования проанализирован интерфейс, предоставляемый LKL, на основе которого была написана утилита для работы с директориями и файлами в разных файловых системах, поддерживаемых ОС Linux, например, EXT4 (тип файловой системы указывается как аргумент).

Изучен принцип работы LKL с дисками на стороне хоста, в ходе чего были обнаружены утечки памяти, что приводило бы к проблемам в случае нескольких дисков. Соответственно, было изменено объединение (*union*) *lkl_disk*: добавлено поле указателя на соответствующую этому диску структуру *virtio_blk_dev* (это позволит корректно удалять диск) и, таким образом, это объединение конвертировано в структуру. Также исправлена функция *lkl_disk_add*[5] и добавлена новая – *lkl_disk_remove* для корректного удаления диска:

```
struct lkl_disk {
    void *dev;
    union {
        int fd;
        void *handle;
    };
};
```

```
void lkl_disk_remove(struct lkl_disk disk)
{
    struct virtio_blk_dev *dev;

    dev = (struct virtio_blk_dev *)disk.dev;
    if (!dev)
        return;

    virtio_dev_cleanup(&dev->dev);
    lkl_host_ops.mem_free(dev);
}
```

Данные исправления в виде ***pull request***[6] приняты разработчиками LKL.

[5] В работе исправление этой функции не приведено

[6] “lkl: Add lkl_disk_remove function”, available from https://github.com/lkl/linux/pull/164
Все поправки этого pull request здесь не представлены (приведены только основные)

Во-вторых, была проанализирована производительность чтения и записи файлов с использованием LKL на примере файловой системы EXT4 (на машине с процессором AMD A10-4600M, 8Gb RAM). Результаты тестов (указано суммарное время открытия файла, прочтения его содержимого и закрытия) приведены на диаграммах 1 и 2.

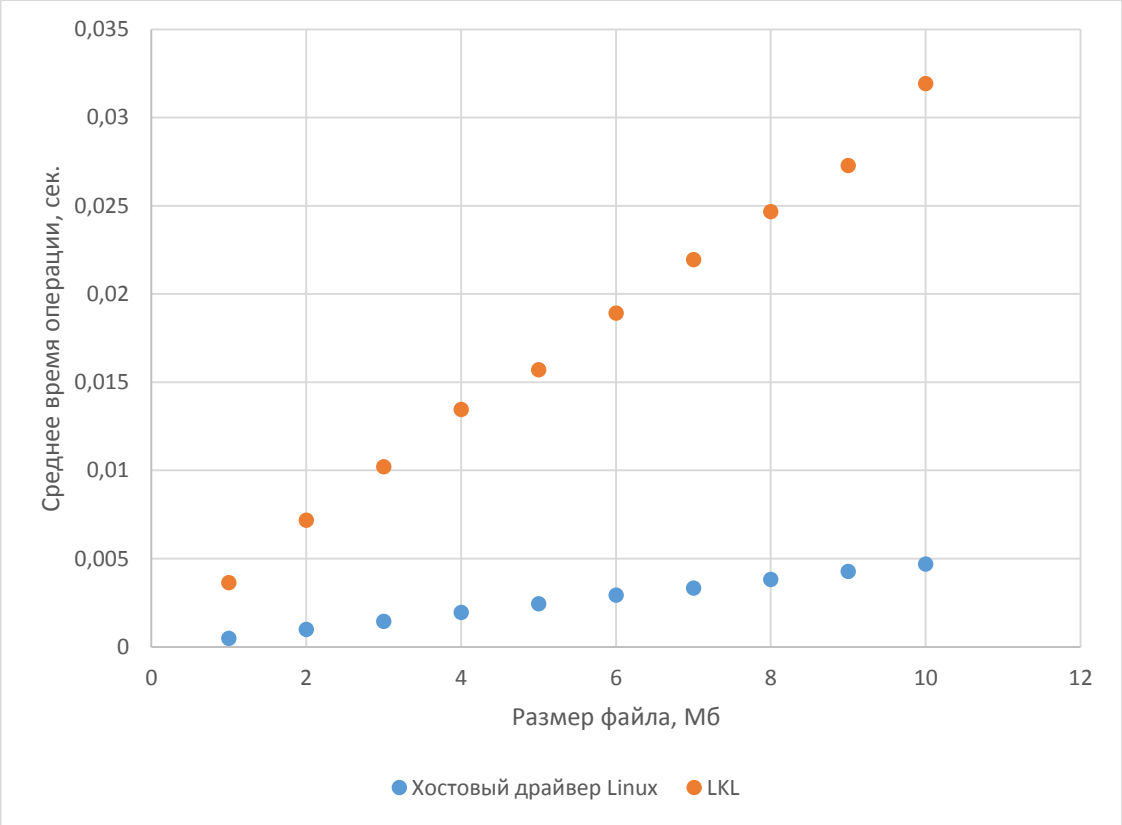

Диаграмма 1. Сравнение времени чтения файла хостовым драйвером Linux и с использованием LKL

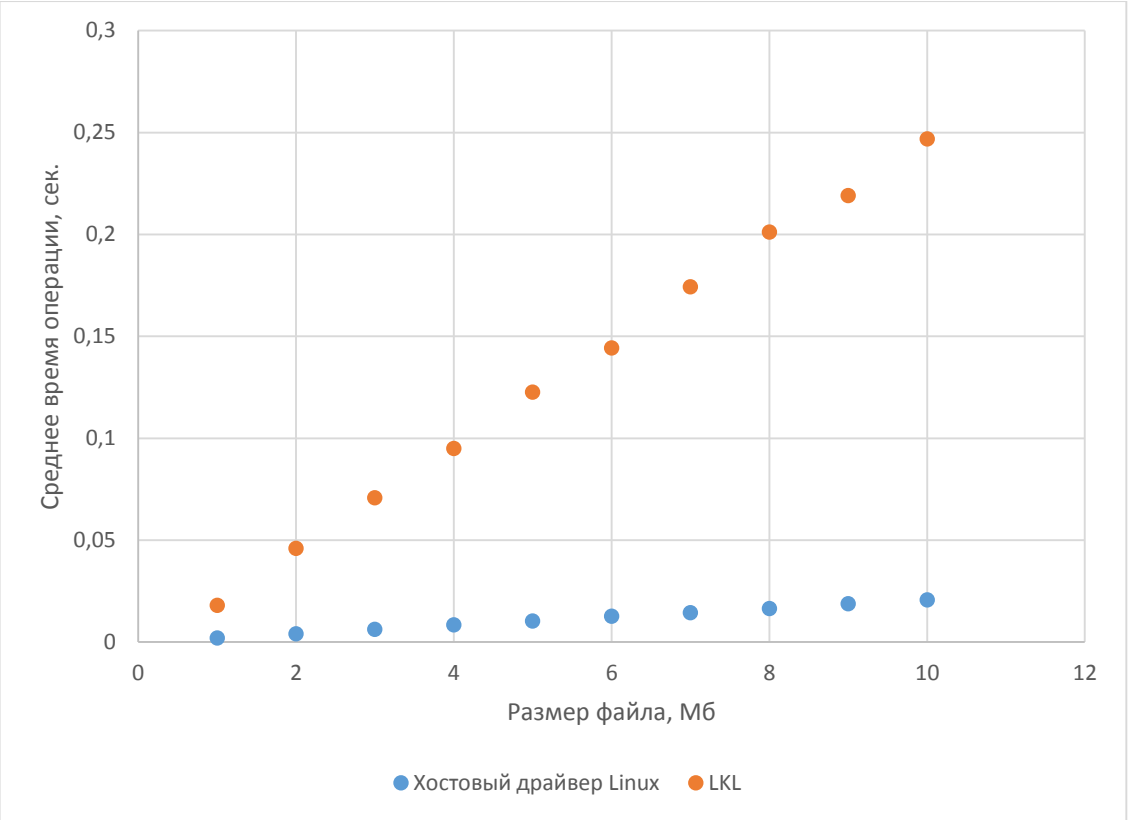

Диаграмма 2. Сравнение времени записи в файл хостовым драйвером Linux и с использованием LKL

Приведенные графики иллюстрируют, что разница времени выполнения операций хостовым драйвером Linux и с использованием LKL на данный момент отличается на порядок.

В-третьих, в LKL интегрирована возможность сбора статистики использования файловой системы.

Для системных вызовов, связанных с файлами и директориями (например, *rename*, *write*, *read* и др.) ведется учет их использования, для чего введена следующая структура:

```
struct fs_using_stats {
    size_t creat_nr;

    size_t write_nr;
    size_t total_write_bytes;
    double avg_write_bytes;


    size_t read_nr;
    size_t total_read_bytes;
    double avg_read_bytes;

    size_t unlink_nr;
    size_t fsync_nr;
    size_t rename_nr;
} fs_stats;
```

Таким образом, можно получить информацию о количестве созданных (*creat_nr*) или удаленных файлов (*unlink_nr*), переименований файлов (*rename_nr*) и др. Также можно узнать среднее количество прочитанных или записанных байт соответствующими системными вызовами *read* и *write*.

Для сбора статистики реализована функция *lkl_log_syscall* (приведем также пример функции для системного вызова *creat*)[7]:

```
static inline void log_creat(long *params) {
    fs_stats.creat_nr++;
}


...

void lkl_log_syscall(long no, long *params) {
    switch (no) {
    case __NR_creat:
        log_creat(params);
        break;
    ...
    }
}
```

Системные вызовы LKL обслуживаются функцией *lkl_syscall* (реализована в *arch/lkl/kernel/syscall.c*), поэтому вызов функции *lkl_log_syscall* для сбора статистики добавлен именно в нее:

[7] Здесь представлена только часть системы сбора статистики (для системного вызова creat)

```
long lkl_syscall(long no, long *params)
{
    struct syscall_thread_data *data = NULL;

#ifdef FS_USING_STATISTICS
    lkl_log_syscall(long no, long *params);
#endif

    ...
}
```

Для получения доступа к структуре *fs_using_stats* извне LKL экспортирована функция:

```
struct fs_using_stats *get_fs_stats(void);
```

Таким образом, предоставляется возможность на стороне хоста получать статистическую информацию об использовании файловой системы.

5. Заключение

В ходе выполнения работы был изучен программный комплекс Linux Kernel Library, исследована возможность использования LKL в работе с различными файловыми системами, поддерживаемыми ОС Linux.

Также были приведены результаты тестов на чтение и запись файлов. На данный момент показатели производительности использования LKL при работе с файлами гораздо ниже, чем у родного драйвера. Хотелось бы в будущем провести детальный анализ узких мест в LKL, где больше всего теряется производительность, и, соответственно, сделать оптимизации, тем самым повышая эффективность LKL.

Кроме того, в LKL интегрирована возможность сбора статистики использования системных вызовов для работы с файлами и директориями. Данная статистика позволит оценить нагрузку на файловую систему, благодаря чему разработчики имеют возможность оптимизировать работу файловой системы. Хотелось бы еще раз отметить, что, более того, LKL позволяет тестировать данные оптимизации без необходимости загружать ядро ОС Linux на физическом сервере или на 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.
5. Таненбаум Э. «Современные операционные системы. 3-е изд.» – СПб.: Питер, 2010 – 1120с. Глава 4.