Правительство Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Санкт-Петербургский государственный университет»

Кафедра Системного Программирования

Дерюгин Денис Евгеньевич

Реализация виртуальной файловой системы для ОС Embox

Бакалаврская работа

Допущена к защите.
Зав. кафедрой:
д. ф.-м. н., профессор Терехов А. Н.

Научный руководитель:
аспирант Козлов А. П.

Рецензент:
главный инженер ООО «ПитерСофтвареХаус» Венгеров В. В.

Санкт-Петербург
2015

SAINT-PETERSBURG STATE UNIVERSITY

Chair of Software Engineering

Denis Deryugin

Embox OS virtual file system implementation

Bachelor's Thesis

Admitted for defence.
Head of the chair:
Professor Andrey Terekhov

Scientific supervisor:
Postgraduate Anton Kozlov

Reviewer:
Chief engineer of PiterSoftwareHouse LLC Victor Vengerov

Saint-Petersburg
2015

Оглавление

Введение

С момента появления файловой организации данных в силу её удобства постоянно появляются новые файловые системы: специализированные, системы общего назначения, сетевые, псевдо-файловые системы, не имеющие представления на физическом носителе.

С одной стороны, это разнообразие открывает большие возможности, но с другой — каждая система имеет свою архитектуру и как следствие — свой интерфейс, что затрудняет их использование приложениями. Для обеспечения переносимости необходимо предоставить некоторую абстракцию для приложений с единым интерфейсом доступа к файлам на разных носителях (рисунок 1). В UNIX-подобных ОС этот механизм обычно называется виртуальной файловой системой (Virtual File System), или ВФС. Впрочем, в области разработки ОС до сих пор нет устоявшегося термина, поэтому можно встретить следующие названия: общая файловая система (Generic File System) в Ultrix, виртуальный переключатель файловых систем (Virtual Filesystem Switch) в System V, устанавливаемая файловая система (Installable File System) в OS/2 и семействе Microsoft Windows.

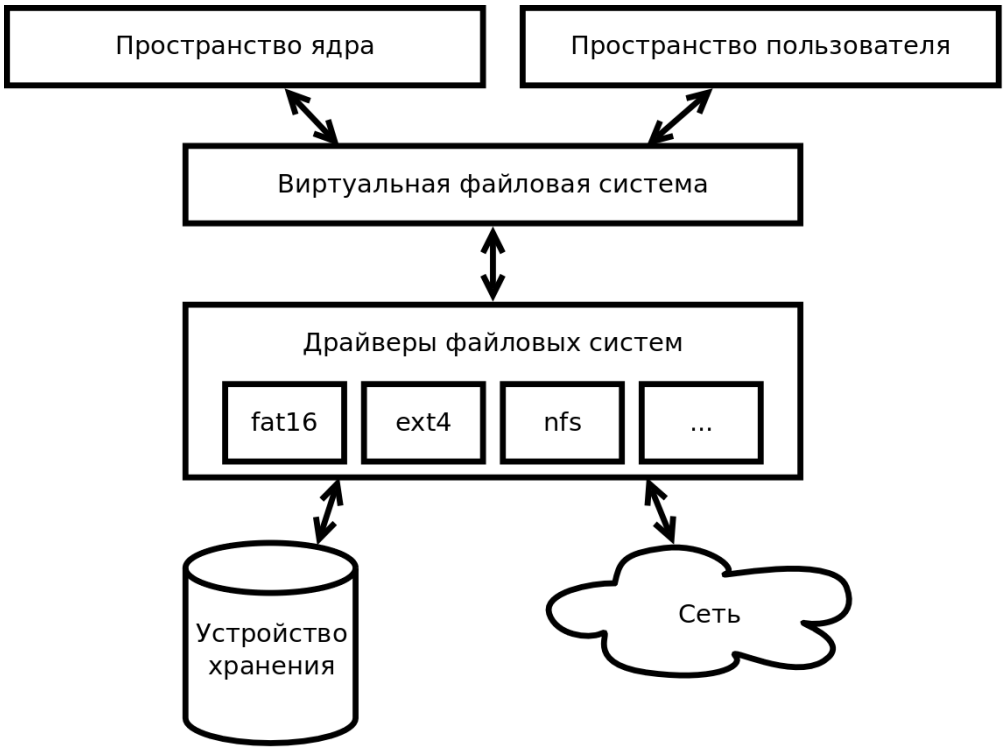

Рис. 1: Слой виртуальной файловой системы и работа с файлами

С помощью этого слоя абстракции приложения могут использовать разные файловые системы через один и тот же интерфейс. Например, виртуальная файловая система может предоставлять приложению единообразный доступ и к дисковой, и к

сетевой файловым системам. Более того, при таком подходе легче переносить и реализовывать драйвера новых файловых систем, так как виртуальная файловая система реализует значительную часть функционала, необходимую для работы с файлами.

Сегодня виртуальные файловые системы повсеместно применяются в операционных системах общего назначения.

Разработка ПО для встроенных систем подразумевает ограничение аппаратных ресурсов. Например, устройство может не иметь внешних носителей данных с большим объёмом памяти (жёсткие диски, флэш-накопители), но, тем не менее, специализированные файловые системы могут быть использованы для сохранения настроек либо для ведения журнала. В то же время, перенос соответствующих механизмов из систем общего назначения порой невозможен из-за аппаратных ограничений. Следовательно, разработчики бывают вынуждены использовать интерфейс конкретной файловой системы или просто работать с памятью без абстракции файловой системы.

Для решения данной проблемы можно использовать часть опыта операционных систем общего назначения, разработав виртуальную файловую систему с учётом указанных выше особенностей.

Практическая часть данной работы осуществлялась на базе ОС Embox[5]. Embox — это проект разработки операционной системы для встроенных систем, развивающийся при поддержке кафедры системного программирования математико-механического факультета Санкт-Петербургского Государственного Университета. В данный момент Embox поддерживает ряд платформ (x86, ARM, MIPS и другие) и, что более важно для виртуальной файловой системы, использует различные файловые системы, как общего назначения (FAT, ext2/3/4, NTFS, NFS) так и специализированные (JFFS2, devfs, ramfs). Виртуальная файловая система, имеющаяся в ОС Embox на данный момент, имеет ряд ограничений, что не позволяет её применять в ряде случаев, например, использовать её на платформах с малым количеством оперативной памяти.

1. Постановка задачи

В рамках данной работы требовалось:

- Разработать архитектуру и реализовать виртуальную файловую систему для ОС Embox с учётом применимости для встроенных систем.
- Реализовать специализированную файловую систему для работы с малым объёмом флэш-памяти.
- Провести апробацию на плате STM32F4Discovery, имеющую 192 КиБ RAM и 1 МиБ флэш-памяти.

2. Обзор

2.1. Применение ВФС во встроенных системах

При разработке операционных систем для встроенных решений чаще всего применяются два подхода.

Один из них подразумевает предоставление базового набора системных вызовов для управления аппаратурой, и главную роль в управлении системой в целом играет пользовательское приложение. В этом случае виртуальная файловая система отсутствует, и пользователь вынужден работать с драйвером той или иной файловой системы непосредственно. Примером такой ОС может послужить freeRTOS[13].

Второй подход — это перенос операционной системы общего назначения. Примером может послужить применение Linux во встроенных системах[3]. Ключевые преимущества данного подхода — бóльшая функциональность и возможность переиспользования кода (например, библиотек и утилит), написанного для системы общего назначения. Стоит учитывать, что так как ОС изначально не разрабатывалась для встроенных систем, аппаратные требования могут быть неадекватны для ряда плат. Например, Pictotux[10], один из самых маленьких компьютеров, управляемых Linux, имеет 8 МиБ SDRAM, что значительно превышает объём памяти упомянутой платы STM32F4Discovery.

2.2. Существующие модели ВФС

Виртуальные файловые системы в том или ином виде начали появляться в 60-х годах двадцатого века. Одна из самых старых моделей впервые была реализована в 1967-м году в системе CP/CMS компании IBM[1] (рисунок 2). Подход заключался

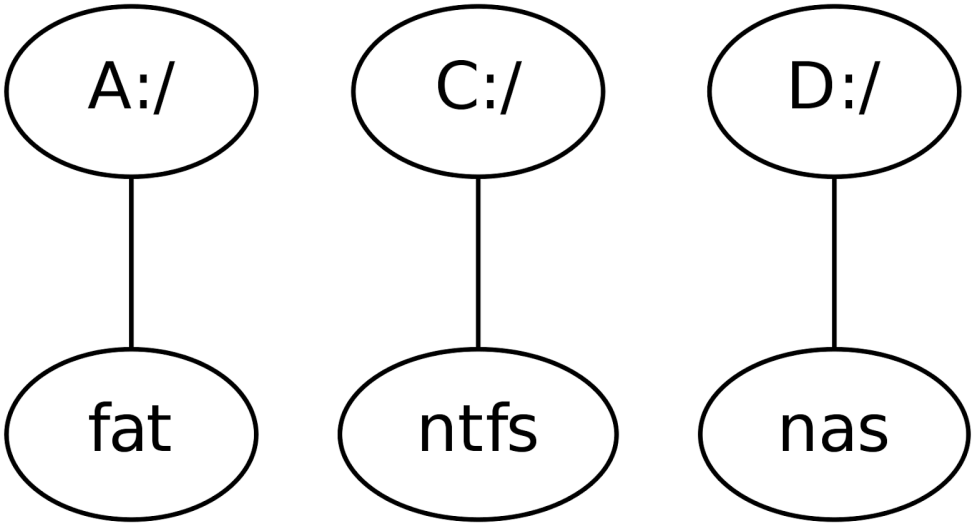

Рис. 2: Модель ВФС в ОС CP/CMS

в присвоении каждому носителю с файловой системой литеры от A до Z, которая приписывалась к началу путевого имени файла и использовалась для определения файловой системы, к которой данный файл относится. На ранних этапах данная модель применялась для файловых систем без иерархической организации файлов с помощью директорий, и так как количество файлов во всей системе редко превышало несколько сотен, разбиение всего файлового пространства на 26 литер позволяло организовать эффективную работу с носителями. Развиваясь, данный подход получил своё применение в таких операционных системах, как MSDOS/Windows, Symbian OS, OS/2 и Atari TOS.

Полноценный механизм монтирования файловых систем в одну иерархическую структуру (рисунок 3) был впервые описан и реализован на базе SunOS в 1985-м году, и в дальнейшем активно использовался при разработке UNIX-подобных систем. Уже тогда применялись не только файловые системы общего назначения, но и псевдофайловые системы, в частности — для отображения информации о состоянии процессов.

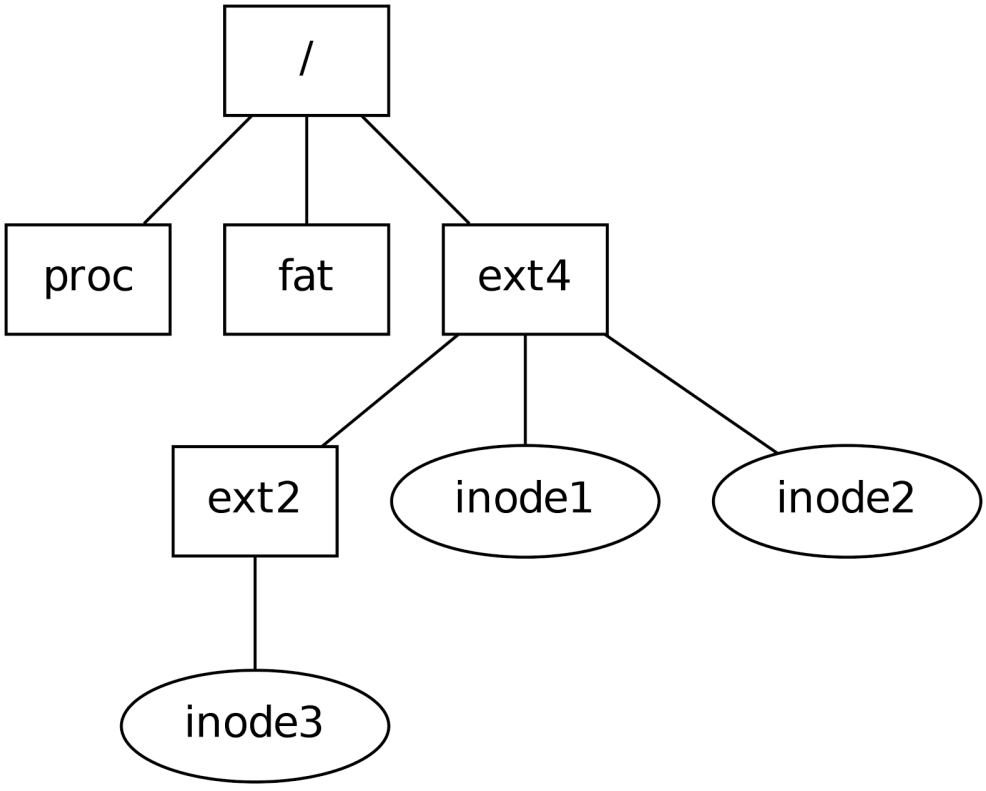

Рис. 3: Модель виртуальной файловой системы в UNIX

Суть данного подхода заключается в построении древовидной структуры, корень которой принадлежит корневой псевдо-файловой системе (*rootvfs*), предоставляющей доступ к другим файловым системам. Каждый узел данной структуры (*vnode*) может быть либо объектом некоторой файловой системы (регулярным файлом, директори-

ей, каналом, символической ссылкой и так далее), либо точкой монтирования файловой системы, которая выглядит как директория и предоставляет доступ к носителю. Специальными системными вызовами можно создавать узлы, монтировать файловые системы и применять привычные для файлов операции.

Подход, описанный выше, продолжает развиваться, и сегодня его идеи используются в UNIX и ряде UNIX-подобных систем с различными дополнениями [4][12]. Так, например, в Linux возможно монтирование одной и той же системы на несколько узлов (даже на узлы внутри самой себя)[7].

Следует отметить, что использование этого подхода ведёт к накладным расходам оперативной памяти для поддержания древовидной структуры. Следовательно, при работе с большим количеством узлов может потребоваться большое количество памяти.

Упрощённым, но менее эффективным подходом к реализации виртуальной файловой системы является хранение списка всех точек монтирования. Тогда при обработке путевого имени файла ВФС пытается определить по префиксу, к какой файловой системе относится указанный файл. После этого остаток пути передаётся драйверу нужной файловой системы. Это приводит к тому, что драйвер файловой системы берёт на себя разбор и интерпретацию пути, и, следовательно, приходится дублировать код, что ведёт к появлению ошибок. Также становится проблематично работать с большим количеством файловых систем, так как время поиска может значительно увеличиться. Для приложений при этом файловая система выглядит как в UNIX-системах, но на самом деле обработка происходит так же, как при назначении литер.

В таблице 1 приведено краткое сравнение трёх указанных подходов.

	Модель представления		
	Дерево	Список	Литера носителя
Простота реализации		+	+
Большое количество ФС	+	+	
Скорость поиска ФС	+		+
Переиспользование кода	+		
Накладные расходы		+	+

Таблица 1: Сравнение моделей ВФС

2.3. Ограничения существующей виртуальной файловой системы в ОС Embox

На данный момент в ОС Embox уже существует виртуальная файловая система, которая позволяет монтировать различные файловые системы и обращаться к файлам через POSIX- и LibC-интерфейсы.

Впрочем, она имеет ряд ограничений, усложняющих её использование на платах с маленьким объёмом оперативной памяти (например — на упомянутой STM32F4Discovery).

К основным ограничениям можно отнести следующие:

- При монтировании файловой системы создаются узлы сразу для всех файлов и директорий, имеющихся на носителе. Это приводит к чрезмерному использованию оперативной памяти, а при большом количестве файлов — к невозможности смонтировать файловую систему в принципе.
- Не хватает некоторых абстракций.
  - Нет абстракции суперблока (*superblock*) или аналогичного по функциям объекта, и, в результате, некоторая информация обрабатывается по-своему в каждом драйвере.
  - Узлы дерева содержат как информацию о файловой иерархии, так и данные, нужные для работы драйвера и для работы с файловыми операциями. Это приводит к тому, что нет чёткого разделения функций виртуальной файловой системы и драйверов файловых систем. Один из примеров неприятных последствий этого — текущая позиция в файле изменяется на уровне драйверов.
- В проекте используется система сборки *Mybuild*[8], что позволяет разбивать исходный код на модули. В настоящее время код многих модулей ядра зависит от кода файловой системы там, где этого можно избежать. Это приводит к проблемам при попытке сконфигурировать систему без использования файловых абстракций (это может быть полезно, если система будет функционировать на платах без внешней памяти) — размер образа становится слишком большим, занимается лишняя оперативная память из-за сегмента bss.

3. Описание решения

Прежде, чем приступать к непосредственной разработке виртуальной файловой системы, стоит рассмотреть возможность переноса аналогичного механизма из какой-либо ОС с открытым исходным кодом. Дело в том, что виртуальная файловая система использует множество различных модулей, а также множество системных модулей зависит от виртуальной файловой системы.

Таким образом, при переносе придётся не только работать непосредственно с логикой файловой системы, но также реализовывать множество преобразующих оборачивающих функций для уже существующих аналогов и переносить множество вспомогательных вызовов целиком при отсутствии аналогичных в целевой системе. Такой процесс не только трудоёмок, но и приводит к увеличению накладных расходов, ведь чем больше модулей переносится из системы общего назначения, тем более громоздкой становится система в целом. Таким образом, разработка файловой системы "с нуля" является более подходящим решением.

3.1. Разработка архитектуры виртуальной файловой системы

Задача виртуальной файловой системы — согласовать интерфейс доступа к файлам (в данном случае — POSIX) с работой соответствующего драйвера. Роль виртуальной файловой системы (вне зависимости от конкретной архитектуры) схематически изображена на рисунке 4.

Так как архитектура виртуальной файловой системы из UNIX является более гибкой и универсальной, именно она была выбрана в качестве основы.

Таким же образом при работе с файловыми системами строится древовидная структура.

Впрочем, сохранение архитектуры в изначальном виде привело бы к чрезмерному использованию ресурсов, поэтому она была упрощена.

Одно из них — отказ от *rootvfs* — корневой псевдофайловой системы. Вместо неё можно использовать определённую файловую систему, заданную в конфигурации. Для систем общего назначения такой подход неудобен, так как для полноценных компьютеров вполне возможно отключение любого носителя во время работы. Следовательно, такой подход лишит возможности отключать носитель с корневой файловой системы без перезапуска системы. В то же время, для встроенных систем характерно наличие неизвлекаемых носителей (например, несколько банков флэш-памяти), которые используются для хранения настроек и ведения журнала, и такие носители как раз могут быть использованы в качестве корневой файловой системы.

Также в UNIX-системах механизм монтирования файловых систем может быть очень запутанным[2]: файловая система может быть смонтирована не только на *rootvfs*, но и на другую ФС, и даже на саму себя. Несколько файловых систем могут быть

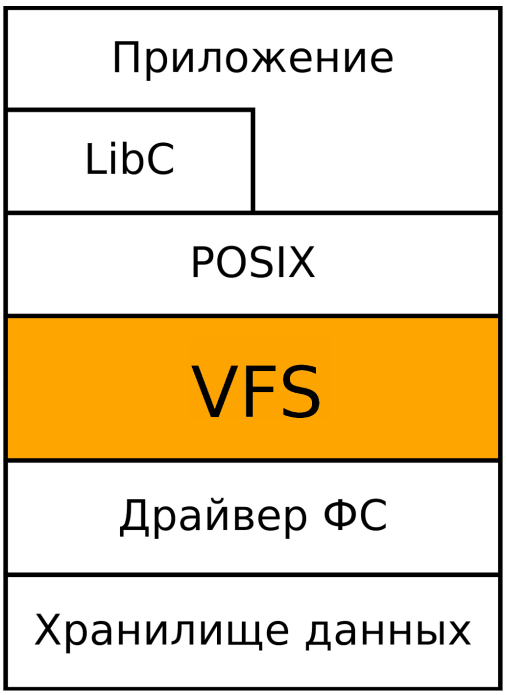

Рис. 4: Роль виртуальной файловой системы при обращении приложения к файлу на носителе

последовательно смонтированы на одну и ту же директорию, образуя стек. А так как для системы общего назначения характерно использование большого числа файловых систем, сложно вести учёт всех точек монтирования. Для этих целей строится дополнительное дерево, содержащее соответствующую информацию. Так как для встроенных систем не характерна такая сложность, можно ограничиться одним файловым деревом, а необходимые для монтирования и демонтирования данные могут быть сохранены в суперблоке файловой системы.

3.1.1. Уровни абстракции

Для доступа к файлам должны быть задействованы четыре следующих механизма:

- Файловый интерфейс. Это могут быть вызовы POSIX либо функции из стандартной библиотеки ввода-вывода.
- Иерархическая структура файлового дерева для поиска файла по путевому имени.
- Драйвер файловой системы для реализации её логики.
- Драйвер блочного устройства для доступа к носителю.

Для разделения логики каждого из данных механизмов выделяются соответствующие абстракции (рисунок 5).

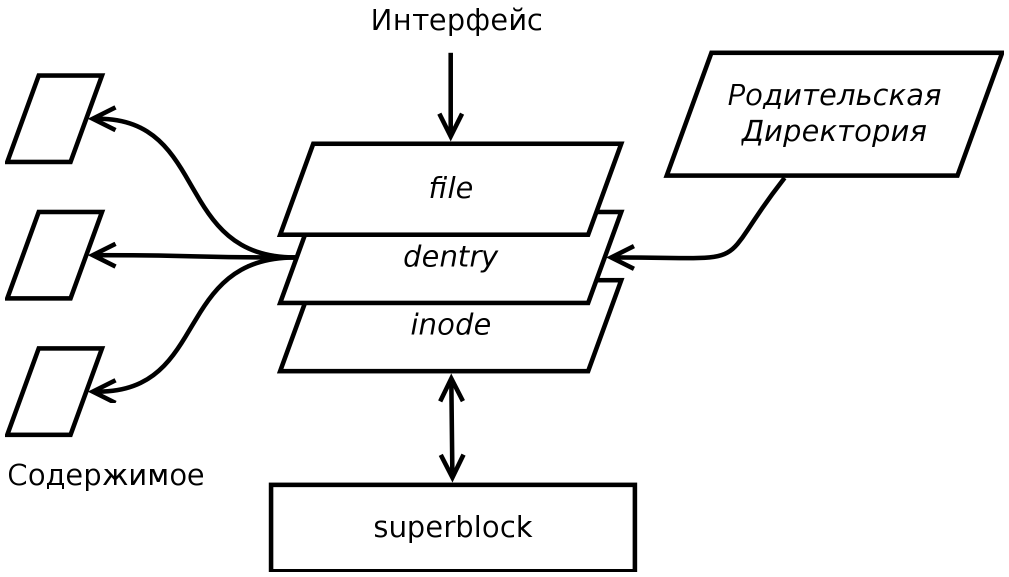

Рис. 5: Абстракции виртуальной файловой системы.

Файлы

Для выполнения таких операций над файлом, как чтение, запись, изменение атрибутов и так далее, используются файлы (*struct file*). Непосредственно с ними приложения не взаимодействуют.

На данный момент в ОС Embox интерфейс stdio из LibC был реализован как обёртка над POSIX. Следовательно, ВФС взаимодействует лишь с POSIX и драйверами файловых систем.

POSIX предоставляет два способа обращения к файлам: через его путевое имя и через номер файлового дескриптора. Соответственно, необходимо уметь находить файл на носителе по его путевому имени, а так же сохранять открытые файлы в ресурсах процесса.

Для получения целочисленного значения файловых дескрипторов для каждого процесса создаётся ресурс — таблица дескрипторов, в которой и сохраняется соответствующая структура.

Структура файла хранит текущую позицию в файле, указатели на запись директории и на узел, связанных с файлом, а также указатели на функции для работы с файлом (*read*, *write*, *open*, *close*, *ioctl*). Данные функции нужны для таких случаев, как, например, запись в сокеты, или вывод в терминал. Для регулярных файлов и директорий они не задаются, и вместо них используются соответствующие функции драйвера файловой системы.

Записи директорий

Иерархическая структура формируется из записей директорий (*struct dentry*) — это структуры, которые описывают содержимое директорий: имя файла, список его

дочерних файлов (если это директория), а так же запись родительской директории. Для каждого процесса важны две записи директорий — корневая и рабочая. С помощью них определяются абсолютный и относительный пути соответственно. Стоит отметить, что не обязательно хранить записи директорий для всей файловой системы.

Достаточный набор для работы ВФС состоит из следующих записей:

- Все открытые файлы.
- Корневые и рабочие каталоги каждого процесса.
- Точки монтирования файловых систем.

Можно не хранить записи, связанные с точками монтирования, но в этом случае придётся заводить дополнительные структуры данных для хранения путей точек монтирования, что не избавляет от расходов оперативной памяти.

Для повышения производительности возможно использование части оперативной памяти в качестве кэша записей директорий.

Узлы

Узлы виртуальной файловой системы (*struct inode*) используются для работы драйвера с файлами на носителе. Стандарт POSIX определяет минимальный набор атрибутов, которые должен иметь файл, и именно в узлах хранится соответствующая информация.

Среди данных содержатся номер узла, права доступа, время создания, позицию на физическом носителе, информацию о типе файла (это может быть обычный файл, директория, канал (*pipe*), символическая ссылка и так далее), специфичную для файловой системы информацию (для FAT, например, это информация о кластерах) и так далее.

3.1.2. Представление файловых систем

Несмотря на всё разнообразие файловых систем, для многих из них можно выделить не только общие метаданные, но и операции над ними.

Каждую файловую систему представляет суперблок (*struct superblock*). Как и в UNIX, в нём содержатся тип файловой системы, размер носителя и информация о прочих структурах для метаданных. Для таких сложных ядер, как Linux, возможно монтирование одной и той же файловой системы на несколько узлов. В этом случае будет использован один суперблок для всех точек монтирования. Для встроенных систем нет нужды в такой возможности, поэтому в данной виртуальной файловой системе, помимо привычных функций, суперблок берёт на себя часть функций дескрипторов монтирования, а именно — сведения о корневой записи директории и информацию о физическом носителе.

3.2. Операции

3.2.1. Монтирование файловых систем

При запуске операционной системы изначально доступна только одна ФС — корневая, которая задаётся в конфигурации при сборке образа системы. Для того, чтобы получить доступ к другим файловым системам, необходимо создать в директорию и смонтировать на неё необходимую файловую систему с помощью системного вызова *vfs_mount()*. В качестве параметров необходимо передать блочное устройство, путевое имя точки монтирования и тип файловой системы.

Для того, чтобы подготовить систему к монтированию, необходимо выделить и инициализировать суперблок в соответствии с содержимым носителя. После этого можно приступать к присоединению файловой системы к общей иерархической структуре.

Сначала производится поиск точки монтирования. Если соответствующая директория найдена, необходимо удалить её из древовидной структуры, так как теперь по путевому имени доступ к ней осуществлён быть не может. Содержимое директории видно только тем процессам, для которых она являлась корневой либо рабочей. Файлы, открытые ранее, будут при этом доступны для записи и чтения.

После этого необходимо выделить новую запись директории и новый узел, которые станут корнем новой файловой системы. Запись размещается в файловом дереве, а у суперблока обновляется корневая запись директории.

При этом необходимо сохранить информацию о директории, на которую производилось монтирование. При демонтировании новой файловой системы нужно будет произвести присоединение прежней директории в тот же самый узел файлового дерева.

3.2.2. Поиск по путевому имени

Когда процесс взаимодействует с файлом, он передаёт путевое имя какому-либо системному вызову, например *open()*, *remove()* или *rename()*. Для того, чтобы понять, к какому файлу на носителе относится данное путевое имя, необходимо разбить полное имя на последовательность вложенных директорий.

Если путевое имя файла начинается с прямого слеша (“/”), то имя является абсолютным, и начальная директория должна быть корневой директорией процесса. Иначе — текущей рабочей директорей.

Пока имя файла не будет полностью разобрано, производится выборка очередной директории с помощью функции *vfs_path_walk()*. При помощи *vfs_lookup()* производится проверка, есть ли директория с таким именем в текущем каталоге. Для данной функции необходима соответствующая поддержка со стороны драйвера файловой системы — по узлу директории необходимо уметь определять, есть ли внутри неё файл

с таким именем. Если есть, то процесс продолжается рекурсивно, иначе — возвращается сообщение об ошибке.

Если поиск файла прошёл успешно, то в оперативной памяти появятся все записи директорий и узлы, соответствующие каждому фрагменту пути от корня до искомого файла. При этом запись искомого файла возвращается вызывающей функции, которая работает с уже выделенными и инициализированными *inode* и *dentry*.

3.2.3. Реализация системных вызовов

Большинство системных вызовов ВФС сначала производят поиск узла по путевому имени и затем реализует необходимую логику.

Та часть вызовов, которая связана с внесением изменений в данные на носителе (read/write/remove/create), вызывает соответствующие функции, реализованные в том или ином драйвере, при необходимости инициализируя узлы и записи директорий и модифицируя соответствующим образом файловое дерево.

Те вызовы, которые не меняют данных на носителе (*open()*, *path_ walk()*, *lookup()*, *iterate()*, *stat()* и так далее), реализуют соответствующую логику, работая с состоянием узлов, записей директорий и файлов.

POSIX-вызовы, в свою очередь, используют вызовы ВФС для реализации своей логики (например, преобразование структуры файла в номер файлового дескриптора, рисунок 6) и генерации кодов ошибок. Для работы с файлами POSIX использует ресурсы процесса, выделяемые при создании: таблица файловых дескрипторов, корневая и рабочая запись процессов.

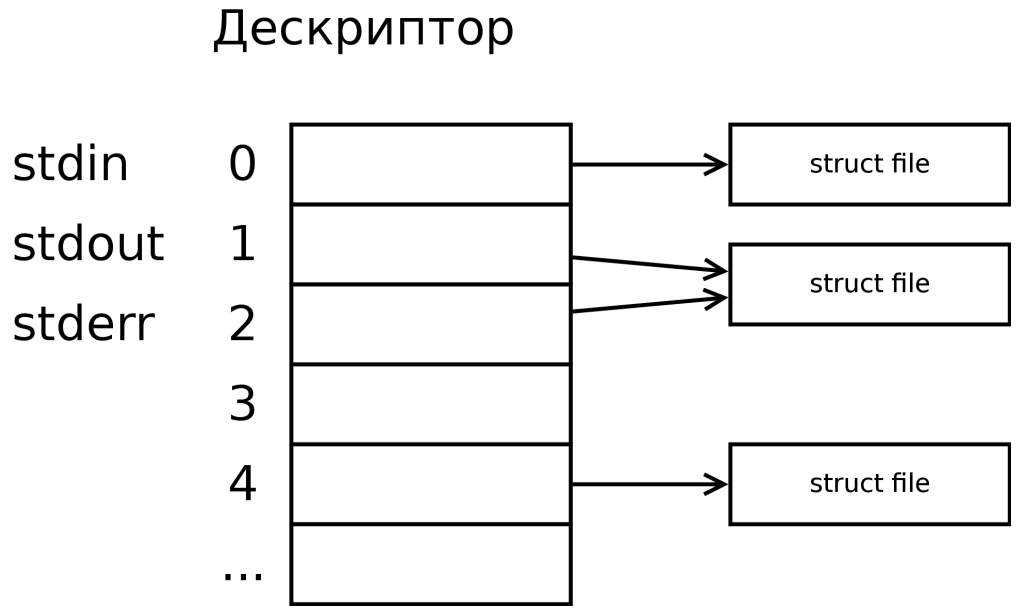

Рис. 6: Сопоставление файла и номера дескриптора.

3.3. Интерфейс драйвера файловой системы

Драйвер взаимодействует лишь с узлами и суперблоком, минуя обработку файловых дескрипторов и записей директорий. Выше уже упоминались некоторые функции, которые должны быть реализованы драйвером файловой системы. Помимо них нужны также:

- Операции суперблока. К ним относятся выделение, освобождение узлов и операции, связанные с монтированием файловой системы.
- Операции узлов. К ним относятся чтение, запись, создание и удаление файлов, создание и удаление директорий, поиск записи по имени в определённой директории, изменение длины файла.

В специальной структуре (*struct fs_ driver*) содержится информация об этих операциях, о типе файловой системы, о том, как инициализировать суперблок соответствующей ФС. Для того, чтобы драйвер мог быть использован ВФС, необходимо его зарегистрировать.

Если какие-то функции не указаны, то при использовании драйвера будут использованы функции по умолчанию. Так, например, суперблок файловой системы на диске может не требовать дополнительной инициализации, в то время как в случае сетевой файловой системы может потребоваться инициализация сетевого соединения.

4. Разработка специализированной файловой системы

Как уже упоминалось во введении, во встроенных системах полноценная файловая система (такая, как ext2 или fat) может быть избыточна. Часто устройству достаточно иметь возможность сохранять настройки и вести журнал.

В данном случае требовалось хранить на флэш-памяти платы STM32F4Discovery настройки светодиодов и сети. Для этого достаточно иметь три файла фиксированной длины в корневом каталоге файловой системы.

Данная плата имеет 1 МиБ встроенной флэш памяти, которая используется для сохранения образа операционной системы. Память делится на несколько ячеек разного размера: четыре блока по 16 КиБ, один блок 64 КиБ и семь блоков по 128 КиБ. Скорее всего, образ системы не займёт все эти блоки, и останется свободная память, которую можно использовать для указанных выше целей.

Существуют специализированные файловые системы, предназначенные для работы с флэш-памятью[9]. Одна из них — JFFS2. Она не подойдёт в данной ситуации, так как оперирует с блоками фиксированной длины (следовательно — 128 КиБ), и ей необходимо хотя бы три блока для перезаписи, что составит более трети всей флэш-памяти. UBIFS[6] также резервирует два блока для информации о томе, один блок для износа ячеек и один блок для чтения и записи логических блоков. В данном случае накладные расходы также слишком велики.

Следовательно, использование таких файловых систем в данном случае не предоставляется возможным. Из схожих соображений, не получится использовать любую журналируемую файловую систему.

Было принято решение разработать простую файловую систему с ограниченным набором возможностей, с поддержкой только регулярных файлов, без прав доступа и с фиксированной длиной файлов.

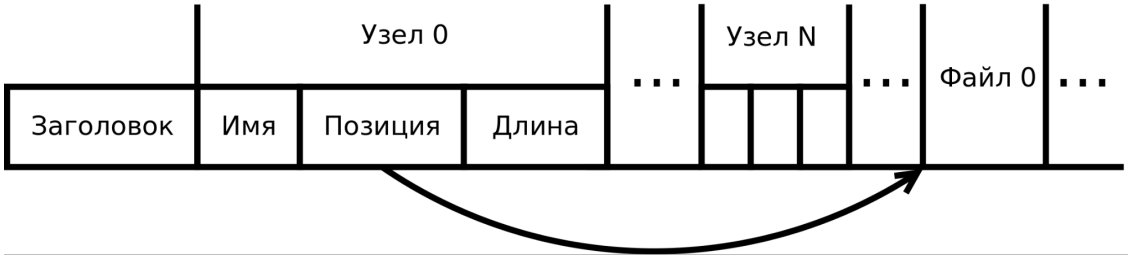

Рис. 7: Логическое устройство файловой системы

Файловая система имеет следующее логическое устройство (рисунок 7): в начале первого блока располагается заголовок файловой системы, содержащий информацию о количестве файлов, о используемых блоках и о свободном месте. После заголовка

идёт список узлов, которые содержат информацию о файлах: название, положение на носителе, размер и флаги.

Как и для всякой файловой системы, работающей с NAND-флэш, имеется проблема перезаписи данных. Записывать данные можно ячейками по 8 байт, а вот затереть данные можно только во всём блоке полностью. Значит, при доступе к меньшему объёму данных, необходимо буферизировать те данные, которые при записи не изменятся. Так как размер даже самого небольшого блока составляет одну двенадцатую объёма всей оперативной памяти, буферизировать перезапись флэш-памяти придётся с использованием некоторого выделенного заранее блока.

Затирание всех блоков длится около 7,94 секунд[11], или, соответственно, около секунды для блока размером 128 КиБ. В это же время, запись 1 МиБ флэш-памяти занимает 1,5 секунды. Следовательно, возникает “бутылочное горлышко”, значительно замедляющее работу с флэш-памятью, особенно при записи данных малыми кусками.

Для ускорения перезаписи был использован следующий механизм. Драйвер файловой системы содержит информацию о том, какой блок копировался в буфер в последний раз. С помощью битового массива хранится информация о том, какие участки буфера уже записаны, а какие нет. Таким образом, если необходимо записать данные в другой блок или перезаписать уже изменённые данные в том же блоке, необходимо скопировать содержимое буфера в нужный блок, затем затереть буфер и обнулить битовый массив.

Данная файловая система была применена для использования первых трёх блоков платы по 16 КиБ.

В результате, 1 блок используется для буферизации. Из оставшихся 32 КиБ для хранения служебной информации используется 96 байт (24 для заголовка и по 24 для каждого узла файлов). Настройки, в свою очередь, занимают 1408 байт.

Таким образом, была реализована специализированная файловая система для работы с флэш-памятью, позволяющая работать с файлами фиксированного размера.

5. Апробация

Для определения эффективности использования оперативной памяти по сравнению со старой виртуальной файловой системой система Embox была собрана в различных конфигурациях для старой и новых виртуальных файловых систем для платы STM32F4Discovery:

- Конфигурация с базовым набором системных модулей без поддержки файловой системы
- Предыдущая конфигурация с поддержкой FAT и специализированной файловой системы

Конфигурация	Код	Данные	.bss	Итого
Без поддержки ВФС	148 504	680	44 960	194 144
С использованием предыдущей ВФС	176 916	860	65 920	243 696
С использованием новой ВФС	162 976	968	72 576	236 520
Предыдущая ВФС с поддержкой FAT и ФС для флэш-памяти	202 552	11 436	85 888	289 876
Новая ВФС с поддержкой FAT и ФС для флэш-памяти	179 432	1 288	73 888	254 608

Таблица 2: Размер секций образа в байтах при различных конфигурациях системы

Размеры образов при данных конфигурациях указаны в таблице 5.

Как следует из результатов замеров, без использования драйвера FAT разница составляет примерно 7 килобайт, в то время как при использовании драйвера разница составляет примерно 35 килобайт, большая часть которых составляет разница в размере кода.

Заключение

В ходе выполнения данной работы были получены следующие результаты:

- Разработана виртуальная файловая система для ОС Embox.
- Разработана специализированная файловая система для работы с флэш-памятью малого объёма, использующая интерфейс виртуальной файловой системы.
- Проведены замеры накладных расходов с использованием виртуальной файловой системы и без неё.

Список литературы

[1] Creasy R. J. The Origin of the VM/370 Time-sharing System // IBM Journal of Research and Development, Vol. 25. — 1981.

[2] Daniel P. Bovet Marco Cesati. Understanding the Linux Kerne. — O'Reilly Media, Inc., 2005.

[3] Embedded Debian Project. — 2015. — URL: http://www.emdebian.org/ (online; accessed: 15.04.2015).

[4] A Formal Model of a Virtual Filesystem Switch / Gidon Ernst, Gerhard Schellhorn, Dominik Haneberg et al. // Proceedings Seventh Conference on Systems Software Verification, SSV 2012, Sydney, Australia, 28-30 November 2012. — 2012. — P. 33–45. — URL: http://dx.doi.org/10.4204/EPTCS.102.5.

[5] Github. Домашняя страничка проекта Embox. — 2015. — URL: https://github.com/embox/embox (online; accessed: 08.05.2015).

[6] Hunter Adrian. A Brief Introduction to the Design of UBIFS. — 2008.

[7] Jones M. Tim. Anatomy of the Linux kernel: History and architectural decomposition // IBM developerWorks. — 2007.

[8] Mybuild, система сборки модульных приложений. — 2012. — URL: http://habrahabr.ru/post/144935/ (дата обращения: 5.03.2015).

[9] Namihira Shinji. Evaluation of Flash File Systems for Large NAND Flash Memory // CELF Embedded Linux Conference. — 2009.

[10] Picotux — The Smallest Computer in the World. — 2015. — URL: http://www.picotux.com/ (online; accessed: 15.04.2015).

[11] Tomar Ankur. — STMicroelectronics: Application Note 3990, 2012.

[12] Verification of a Virtual Filesystem Switch / Gidon Ernst, Gerhard Schellhorn, Dominik Haneberg et al. // Verified Software: Theories, Tools, Experiments / Ed. by Ernie Cohen, Andrey Rybalchenko. — Springer Berlin Heidelberg, 2014. — Vol. 8164 of Lecture Notes in Computer Science. — P. 242–261. — URL: http://dx.doi.org/10.1007/978-3-642-54108-7_13.

[13] freeRTOS. — 2015. — URL: http://www.freertos.org/ (online; accessed: 08.04.2015).