

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ
ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования

«ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ
УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ» (ТУСУР)

Кафедра автоматизированных систем управления (АСУ)

ОПЕРАЦИОННЫЕ СИСТЕМЫ

Тема 4. Управление файловыми системами ОС

Учебно-методическое пособие

для студентов уровня основной образовательной программы: **бакалавриат**
направление подготовки: **09.03.01 - Информатика и вычислительная техника**
направление подготовки: **09.03.03 - Прикладная информатика**

Разработчик
доцент кафедры АСУ

В.Г. Резник

Резник В.Г.

Операционные системы. Тема 4. Управление файловыми системами ОС. Учебно-методическое пособие. – Томск, ТУСУР, 2021. – 49 с.

Учебно-методическое пособие предназначено для изучения теоретической части и выполнения лабораторной работы №4 по теме «Управление файловыми системами ОС» учебной дисциплины «Операционные системы» для студентов кафедры АСУ ТУСУР уровня основной образовательной программы бакалавриата направлений подготовки: «09.03.01 - Информатика и вычислительная техника» и «09.03.03 - Прикладная информатика».

Оглавление

Введение.....	4
1 Тема 4. Теоретическая часть.....	5
1.1 Устройства компьютера.....	5
1.1.1 BOOT-сектор и разделы винчестера.....	8
1.1.2 Загрузочные сектора разделов.....	10
1.1.3 Структура файловой системы FAT32 (VFAT).....	12
1.1.4 Структура файловой системы EXT2FS.....	16
1.1.5 Сравнение файловых систем.....	24
1.2 Стандартизация структуры ФС.....	27
1.2.1 Модули и драйверы ОС.....	33
1.2.2 Системные вызовы ОС по управлению устройствами и ФС.....	34
1.2.3 Три концепции работы с блочными устройствами.....	36
1.3 Разделы дисков и работа с ними.....	39
1.3.1 Монтирование и демонтирование устройств.....	40
1.3.2 Файловые системы loopback, squashfs, overlayfs и fuse.....	42
1.3.3 Дисковые квоты.....	46
2 Лабораторная работа №4.....	47
2.1 Типы, имена и узлы устройств.....	47
2.2 Структура винчестера и файловые системы.....	47
2.3 Стандартизация структуры ФС.....	48
2.4 Модули и драйверы ОС.....	48
2.5 Концепции работы с устройствами.....	48
2.6 FUSE и другие специальные ФС.....	48
2.7 Подключение рабочей области пользователя ucrk.....	48
Список использованных источников.....	49

Введение

Данная тема нашей дисциплины посвящена управлению файловыми системами ОС. Поскольку файловые системы состоят из файлов, то управление касается и их. Мы будем рассматривать оба случая предполагая, указанное различие будет понятным из контекста. Перечень изучаемых вопросов и их место в учебном материале дисциплины «Операционные системы» изложен в источнике [1], основным учебником является [2], а дополнительным [3]. В качестве практических задач, которые не могут обойтись без должного уровня знаний файловой системы, мы используем учебный материал, изложенный в [4].

В целом, основные понятия о файлах и файловых системах (ФС) были рассмотрены в предыдущих темах:

- тема 1: подразделы 1.7 и 1.8 в [5];
- тема 2: подразделы 1.5 -- 1.7 в [6];
- тема 3: подразделы 1.6 и 1.7 в [7].

Данная тема полностью раскрывает все вопросы, необходимые для управления ФС, тем самым, обеспечивая студента должным уровнем подготовки.

Последовательность изложения учебного материала ведётся от простого к сложному, одновременно интегрируя уже изученные знания и детализируя их по мере необходимости. Хотя такое утверждение является достаточно субъективным, оно опирается на имеющийся опыт преподавания данной дисциплины.

Первый раздел, озаглавленный «Тема 4. Теоретическая часть», собственно и содержит описание всех заявленных вопросов. Здесь же, кроме теоретического материала, приводятся примеры его конкретного использования. Эти примеры и ложатся в основу лабораторной работы по данной теме.

Второй раздел, озаглавленный «Лабораторная работа №4», содержит методический материал по практическому закреплению полученных знаний. Средой исполнения этих работ является ОС УПК АСУ, установленная в учебных классах кафедры АСУ или на личных компьютерах студентов. Успешно выполненной считается работа описанная в личном отчёте студента и проверенная преподавателем.

1 Тема 4. Теоретическая часть

Данный учебный материал предполагает, что предыдущие темы студентом изучены и закреплены на практике тремя лабораторными работами. Имея полученные знания, можно провести нужный уровень детализации в изучении дисциплины. Здесь, такая детализация проводится для темы «Управление файловыми системами ОС».

1.1 Устройства компьютера

В первую очередь, термин *устройства компьютера* ассоциируются с аппаратурой ЭВМ, что в общем случае так и есть. Но в предмете нашей дисциплины, аппаратура компьютера не является непосредственно доступной для пользователя или программ, работающих в пользовательском режиме. Как ранее было отмечено:

- *доступ к устройствам* ЭВМ возможен только через ядро ОС;
- *все устройства* ЭВМ имеют отображение в виде имён в ФС ОС.

С другой стороны, устройства компьютера являются *ресурсами ОС*, которые управляются ядром ОС.

С третьей стороны, *сами файловые системы ОС* располагаются на устройствах компьютера.

Указанное выше противоречие, заключающееся в том, что устройства отображаются в ФС, которые сами находятся на устройствах, разрешается следующим образом:

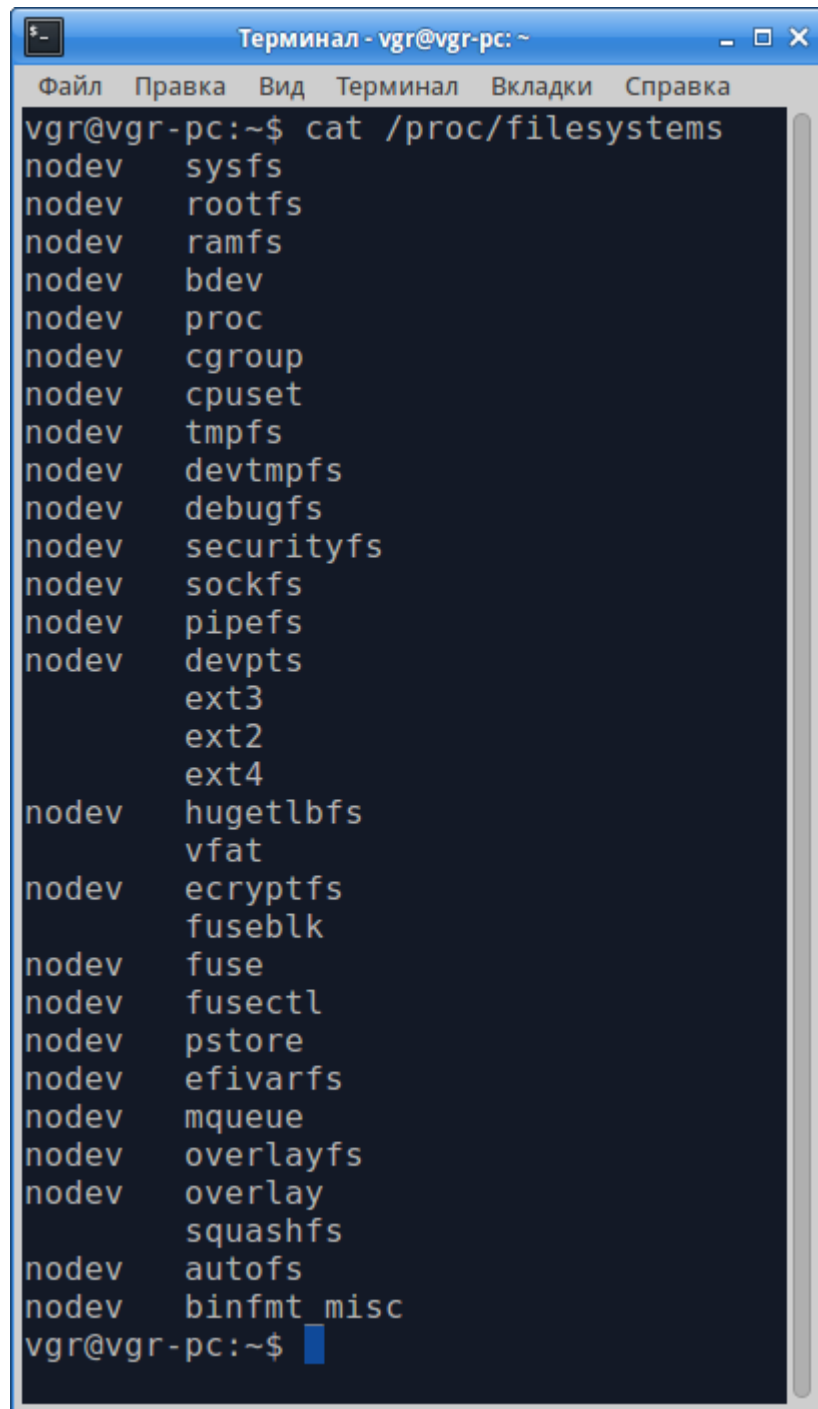
- *все устройства ОС* делятся на *блочные* и *символьные*;
- *блочные устройства ОС* могут содержать файловые системы;
- *символьные устройства ОС* — устройства ОС, не являющиеся блочными;
- *в ядре ОС создаются блочные устройства*, которые не относятся к аппаратным средствам ЭВМ: *псевдоустройства ядра ОС* или устройства *nodev*;
- *псевдоустройства ядра ОС* имеют имена, которые почти все совпадают с именами соответствующих файловых систем; эти имена соответствуют вершинам файловых систем (ФС);
- *псевдоустройство с именем rootfs* является корнем *виртуальной файловой системы* (VFS — *Virtual File System*).

Когда GRUB загрузил ядро ОС и передал ему файл, с временной файловой системой, тогда, после инициации внутренних параметров ядра, проводится *монтирование (подключение) корневой файловой системы*. Это подключение (монтирование) проводится самим ядром ОС и относится к *внутренней точке ядра с именем rootfs*.

Далее, ядро ОС создаёт первый процесс, которым является интерпретатор shell. Он, от имени пользователя *root* (UID=0) и группы GID=-1, начинает выполнять сценарий */init*. Поскольку пользователь *root* работает в режиме пользователя, то он воспринимает точку ядра *rootfs* как */*.

В процессе выполнения сценария */init*, к директориям корневой ФС монтируются другие файловые системы.

На рисунке 1.1 выведены типы файловых систем, поддерживаемых текущим сеансом ОС УПК АСУ. Все *псевдоустройства ядра ОС* обозначены как *nodev*.

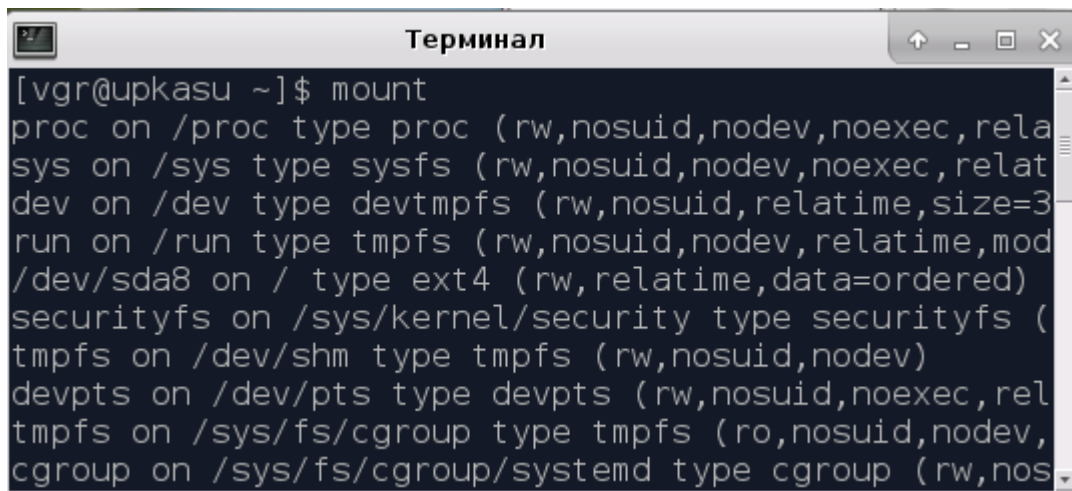


```
Терминал - vgr@vgr-pc: ~
Файл  Правка  Вид  Терминал  Вкладки  Справка
vgr@vgr-pc:~$ cat /proc/filesystems
nodev    sysfs
nodev    rootfs
nodev    ramfs
nodev    bdev
nodev    proc
nodev    cgroup
nodev    cpuset
nodev    tmpfs
nodev    devtmpfs
nodev    debugfs
nodev    securityfs
nodev    sockfs
nodev    pipefs
nodev    devpts
nodev    ext3
nodev    ext2
nodev    ext4
nodev    hugetlbfs
nodev    vfat
nodev    ecryptfs
nodev    fuseblk
nodev    fuse
nodev    fusectl
nodev    pstore
nodev    efivarfs
nodev    mqueue
nodev    overlayfs
nodev    overlay
nodev    squashfs
nodev    autofs
nodev    btrfs
nodev    misc
vgr@vgr-pc:~$
```

Рисунок 1.1 — Список файловых систем ядра ОС УПК АСУ

На рисунке 1.2 представлен вывод части сообщений команды *mount*. Хорошо видно, монтирование некоторых псевдоустройств на директории корневой ФС.

По традиции, все ОС UNIX и Linux *отображают имена файлов устройств* в директорию */dev* корневой файловой системы. Для устройств, точкой монтирования ядра ОС является *dev*, что также хорошо видно из рисунка 1.2.



```
[vgr@upkasu ~]$ mount
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
sys on /sys type sysfs (rw,nosuid,nodev,noexec,relatime)
dev on /dev type devtmpfs (rw,nosuid,relatime,size=32768k)
run on /run type tmpfs (rw,nosuid,nodev,relatime,mode=755)
/dev/sda8 on / type ext4 (rw,relatime,data=ordered)
securityfs on /sys/kernel/security type securityfs (rw,nosuid,nodev,noexec,relatime)
tmpfs on /dev/shm type tmpfs (rw,nosuid,nodev)
devpts on /dev/pts type devpts (rw,nosuid,noexec,relatime)
tmpfs on /sys/fs/cgroup type tmpfs (ro,nosuid,nodev,noexec,relatime)
cgroup on /sys/fs/cgroup/systemd type cgroup (rw,nosuid,nodev,noexec,relatime)
```

Рисунок 1.2 — Примеры монтирования псевдоустройств ядра

Общая семантика слова *устройство* соответствует семантике английского слова *device*. Реальные файловые системы ОС, на которых долговременно хранится информация, находятся на физических устройствах ЭВМ. Поскольку физические устройства типа винчестера разбиваются на разделы, то *реальные ФС создаются в разделах блочных устройств*. Именно разделы блочных устройств *монтируются к директориям VFS*.

В терминологии ядра, имя устройства, которое отображается в директории */dev*, называется *узел (node)*.

NODE - *узел* - специальная *именованная структура*, отображаемая в директории */dev*, создаваемая для связи ядра ОС с физическим устройством ЭВМ.

Замечание

Чтобы ядро ОС могло работать с физическими устройствами ЭВМ, *узлы (node)* должны быть созданы заранее и иметь отображение в директории */dev*.

Узлы создаются как для *блочных*, так и для *символьных устройств*.

Каждый узел имеет *имя, тип, старший_номер и младший_номер*.

Все узлы хранятся в *специальной таблице* (специальной файловой системе). Например, из рисунка 1.2 видно, что директория */dev* монтирована в точку ядра ОС с именем *dev*, имеющей тип файловой системы *devtmpfs*.

Для создания устройств используется утилита **mknod**, в формате;

```
mknod [OPTION]... NAME TYPE MAJOR MINOR
```

где NAME — имя устройства;

TYPE равно **b** для блочного устройства;

TYPE равно **c** для символьного устройства;

TYPE равно **p** для канала FIFO;

MAJOR, MINOR — старший и младший номер устройства.

Например,

```
[ -e /dev/console ] || mknod -m 0600 /dev/console c 5 1  
[ -e /dev/null ] || mknod /dev/null c 1 3
```

В этом примере, для устройства **консоли** и устройства **null**, проверяется наличие узла по его имени, и если узлы отсутствуют, то они создаются.

На рисунке 1.3, с помощью команды **ls -l** и утилиты фильтра **grep**, показан вывод информации об узлах **console** и **null**.

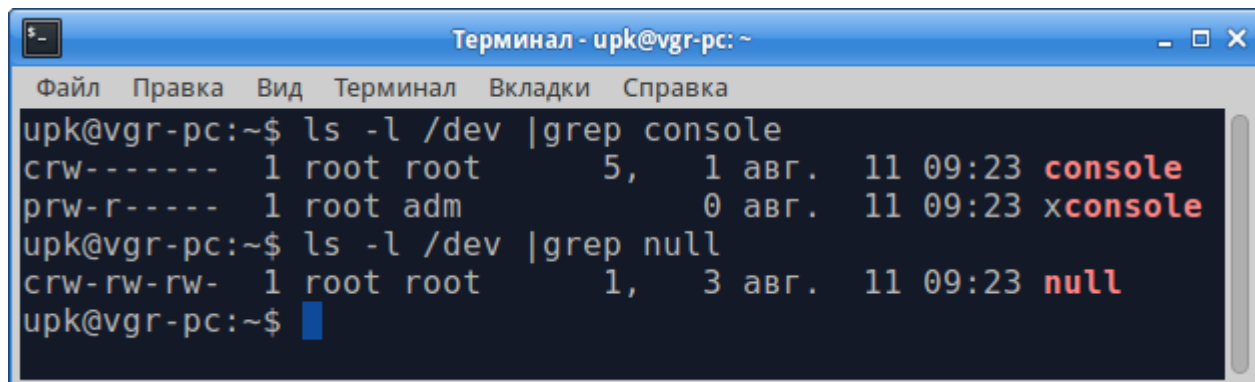


Рисунок 1.3 — Вывод информации об узлах устройств

Замечание

Для создания узлов необходимо знать не только имена и типы устройств, но и их номера (младшие и старшие). Наличие большого количества различных физических устройств ЭВМ и их различная группировка, создаёт ряд известных проблем, при создании и сопровождении узлов. Более подробно, эти проблемы обсуждаются в подразделе 1.10 «Три концепции работы с устройствами».

В данном курсе, подробная работа с символьными устройствами не рассматривается, поскольку она всегда имеет свою специфику и детальное знание самих устройств. Поэтому информация об использовании символьных устройств даётся только в общем контексте их применения в ОС.

1.1.1 BOOT-сектор и разделы винчестера

Учитывая тематику изучаемого материала, мы ограничимся рассмотрением конкретного физического устройства ЭВМ, известного как **винчестер** или «**жёсткий диск**», который является **основным блочным устройством** для долговременного хранения информации и одновременно - **оперативно используется в работе ОС**.

Кроме того, именно на одном из разделов винчестера размещается файловая система, рассматриваемая ядром ОС как **корневая ФС**.

Все блочные устройства, на которых размещаются конкретные ФС имеют свою структуру, определяемую физическим типом устройства. Такую структуру

имеют и физические устройства, называемые **винчестер**. Широкое применение этого оборудования привело к стандартизации его общей структуры, которая уже была рассмотрена ранее в теме 2 [6, подраздел 1.6 «Винчестер и загрузочные устройства»]:

- **классическая** структура MBR;
- **новая структура** GPT.

Широкое и успешное применение применение устройств типа винчестер и их общих структур привело к тому, что эти структуры стали переноситься на другие устройства, физически отличающиеся от винчестера. Примером таких устройств является flashUSB, которое является микросхемой и не имеет дисков, головок и секторов. Тем не менее, перенос структуры винчестера на flashUSB позволяет использовать это устройство как блочное, с наименьшими изменениями в системном ПО ОС.

Ограничиваясь только классической структурой MBR, можно выделить:

- **главный загрузочный сектор** (MBR), который не зависит от типа ОС;
- **загрузочные секторы** (блоки) логических дисков (**разделов**), которые зависят от ОС только в плане поддерживаемых ей типов ФС;
- **специальные области разметки** и **корневой каталог**, зависящие от типа файловой системы;
- **область данных** – файлы и каталоги конкретной файловой системы;
- **цилиндр** для выполнения диагностических операций чтения/записи.

Указанная структура демонстрируется схемой, представленной на рисунке 1.4.

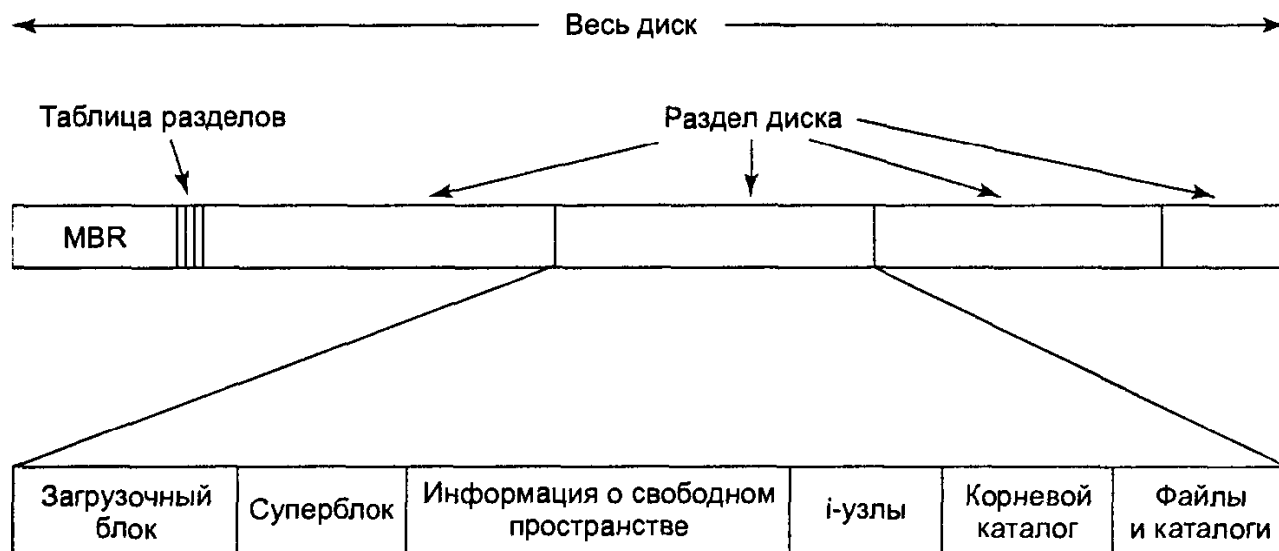


Рисунок 1.4 - Общая структура винчестера (жесткого диска)

Для поддержки такой структуры блочных устройств, каждая ОС имеет специальные **системные утилиты**.

Для примера, в ОС УПК АСУ, такими утилитами являются:

- **fdisk** - позволяет показать все блочные устройства ЭВМ, а также **обеспечить**

- *разбиение устройства* на нужное количество разделов;
- **mkfs** — позволяет *создать файловую систему на заданном разделе* заданного блочного устройства;
- **mknod** — позволяет *создавать узлы в ФС*, которые именуют блочные и символичные устройства для доступа к ним ПО пользовательского режима;
- **mount** — позволяет *монтировать (подключить) конкретную ФС* к древовидной структуре файлов ядра ОС;
- **umount** — позволяет *размонтировать (отключить) конкретную ФС* от древовидной структуры файлов ядра ОС.

Следуя основной парадигме UNIX-систем: «Все есть файл», разработчики ОС создавали *свои файловые системы*, которые *монтировались* при загрузке ОС и использовались, в дальнейшем, как *уникальные* хранилища информации.

Фактически, задача загрузки первых реализаций ОС сводилась (см. рисунок 1.4):

- *к использованию загрузочного блока* для загрузки ядра и передачи ему *информации о разделе конкретного блочного устройства*;
- *ядро ОС*, проведя инициализацию внутренних параметров и внешних устройств ЭВМ, *находила этот раздел* и *монтировала его* как *корневую файловую систему*.

Хотя каждая ФС была оптимизирована под ядро конкретной ОС, подобный подход имел две существенные проблемы:

- *правильное монтирование* корневой ФС было возможным только после предварительной инсталляции ОС, *посредством специального инсталлятора*;
- *перенос данных* с других («не родных») ФС был крайне затруднительным.

Замечание

Такая ситуация вполне устраивала многие корпорации, но **в 1985 году**, компания Sun Microsystems предложила **VFS** — Virtual File System (File System Switch) — *Виртуальную Файловую Систему* для ОС SunOS.

1.1.2 Загрузочные сектора разделов

Для блочного устройства типа винчестер (жёсткий диск), минимальным физически читаемым/записываемым объёмом информации (данных) является сектор, который для современного оборудования ЭВМ равен **512 байт**.

Сектор — физическая единица обмена данными ОС с блочными устройствами.

Файловые системы (ФС), которые стали создаваться для различных нужд ОС, стали использовать *понятие блока*.

Блок — логическая единица обмена данными ОС с блочными устройствами.

Размер блока равен *одному сектору* или *кратен целому числу* таких секторов.

Понятие блока вводится с целью преодоления ограничений, накладываемых способом адресации разделов в таблице MBR и ПО BIOS, что ограничивает ис-

пользуемый объём пространства самих блочных устройств. Например, **четыре байта** адреса MBR не могут адресовать более **двух ТБайт** адресного пространства блочного устройства.

Кластер — это эквивалент понятию блока, используемый корпорацией Microsoft в своих ОС MS DOS и MS Windows.

Другая проблема конструктивных особенностей блочных устройств, таких как винчестер, связана с тем, что линейная скорость движения головки чтения/записи по внешним трекам диска выше, чем по внутренним. Соответственно, на внешних треках можно расположить больше секторов, чем на внутренних секторах. Это позволяет более эффективно использовать поверхность диска, но делает непригодной систему адресации секторов: цилиндр, головка, сектор (**CHS**).

Выход из данной ситуации — использование технологии **LBA** (*Logical Block addressing*), которая перекладывает саму адресацию секторов на контроллер винчестера, а для операционным системам (ОС) предоставляется возможность использования упорядоченной последовательности логических блоков.

Наконец, разработка новой структуры блочных устройств **GUID Partition Table** (GPT) снимает проблему логической адресации больших объёмов данных блочных устройств, *расширив формат указателей адреса и стандартизовав размер LBA на уровне одного сектора (512 байт)*.

Другой круг проблем связан с ПО самой BIOS, рассмотренной нами в [6, разделы 1.2 и 1.3]. Дело в том, что функции и прерывания BIOS могут работать только на уровне секторов и LBA, а также понимают только структуру MBR. По этой причине, для загрузки ОС необходимо в разделе корневой файловой системы *выделить область для ПО загрузчика*, а в программный код первых **446 байт** MBR *прописать адрес загрузчика*. Очевидно, что такой адрес должен указывать на фиксированное местоположение ПО загрузчика относительно начала раздела. Поэтому, практически все ФС включают в свою структуру некоторый блок загрузки, как это показано на рисунке 1.4.

Как следствие, такой подход позволяет загружать только одну ОС с одного блочного устройства.

Альтернативный подход предполагает *смещение начала первого раздела* относительно сектора MBR. Такое смещение позволяет записать сразу же после сектора MBR ПО загрузчика, что и используется нами при установке **ПО GRUB**.

Замечание

Использование альтернативных загрузчиков делает не нужным использование загрузочных секторов разделов, но, в силу преобладания структур ФС, данные сектора остаются.

Очевидно, что структуры различных ФС могут сильно отличаться друг от друга. На рисунке 1.4 приведена обобщённая структура ФС, характерная для ОС UNIX. Далее, на примере двух ФС — **FAT32** и **ext2fs**, мы рассмотрим их основные особенности, а затем проведём их сравнительный анализ.

1.1.3 Структура файловой системы FAT32 (VFAT)

FAT32 - файловая система компании *Microsoft*, разработанная *в августе 1996 года*. Она является преемницей файловых систем *FAT8*, *FAT12* и *FAT16*, начало разработки которых положено Биллом Гейтсом и Марком Мак Дональдом, *в 1976 - 1977 годах*.

VFAT — расширение *FAT*, появившееся в *MS Windows 95* и дающее возможность *использовать имена файлов длиной до 255 символов*, в кодировке *UTF-16LE*.

Фактически, VFAT и используется во всех ОС, как универсальный хранитель информации.

FAT – *File Allocation Table* или *Таблица размещения файлов*.

32 – *число бит*, используемое для нумерации *кластеров* (блоков) раздела файловой системы (на самом деле используется **28 бит**).

Кластер – это объем данных, которыми оперирует файловая система. Обычно, для 8 Гбайт раздела используется 4-х КБайтные кластеры (8 секторов диска).

Один сектор диска – 512 байт.

Загрузочный сектор раздела содержит:

- *блок параметров диска (BPB)*, в котором содержится информация о разделе: размер и количество секторов, размер кластера, метка тома и другие;
- *загрузочный код* – программу, с которой начинается процесс загрузки операционной системы (для MS-DOS и MS Windows-9X – файл *Io.sys*).

На этапе логического форматирования каждого раздела (*логического диска*) создаются четыре логических области:

- *загрузочный сектор* (boot sector);
- *таблица размещения файлов* (FAT1 и FAT2);
- *каталог*;
- *область данных*.

Загрузочный сектор содержит в себе, *кроме кода загрузчика*, *таблицу BPB* и двухбайтовую сигнатуру **55AA** (0xAA55).

BPB — *BIOS Parameter Block* — таблица содержащая множество параметров, определяющих характеристики блочного устройства и самой файловой системы.

Замечание

Для нужд FAT32, таблица BPB была расширена и названа **BF_BPB**.

BF_BPB — *Big FAT BIOS Parameter Block*.

Размер загрузочного сектора:

- *один физический сектор* для FAT16;
- *три физических сектора* для FAT32.

Далее, в таблице 1.1, представлена структура первого сектора (*boot sector*) файловой системы FAT32.

Таблица 1.1 - Первый сектор раздела FAT32

Смещение, байт	Длина поля, байт	Обозначение	Содержание
0x00 (0)	3	<i>JUMP 90</i>	Безусловный переход на начало загрузчика
0x03 (3)	8		<i>Системный идентификатор</i> — начало <i>BF_BPB</i>
0x0B (11)	2	<i>SectSize</i>	Размер сектора, байт (обычно 512)
0x0D (13)	1	<i>ClastSize</i>	Число секторов в кластере
0x0E (14)	2	<i>ResSecs</i>	Число зарезервированных секторов, равно 32
0x10 (16)	1	<i>FATcnt</i>	Число копий FAT, обычно 2
0x11 (17)	2	<i>RootSize</i>	Количество элементов в корневом каталоге
0x13 (19)	2	<i>TotSect</i>	Общее число секторов. 0, если размер больше 32 Мб
0x15 (21)	1	<i>Media</i>	Дескриптор носителя
0x16 (22)	2	<i>FATsize</i>	Количество секторов на элемент таблицы FAT
0x18 (24)	2	<i>TrkSecs</i>	Число секторов на дорожке
0x1A (26)	2	<i>HeadCnt</i>	Число рабочих поверхностей (число головок)
0x1C (28)	4	<i>HidnSecs</i>	Число скрытых секторов, которые расположены перед загрузочным сектором. Используются при загрузке для вычисления абсолютного смещения корневого каталога и данных
0x20 (32)	4		Число секторов на логическом диске
0x24 (36)	4		Число секторов в таблице FAT
0x28 (40)	2		Расширенные флаги
0x2A (42)	2		Версия файловой системы
0x2C (46)	4		Номер кластера для первого кластера каталога. Обычно, значение этого поля равно 2.
0x30 (48)	2		Номер информационного сектора ФС
0x34 (50)	2		Номер сектора с резервной копией загр. сектора
0x34 (52)	12		Зарезервировано FAT32 (<i>расширение BF_BPB</i>)
0x40 (64)	1		Физический номер устройства: 0x00 — <i>флоппи диск</i> ; 0x80 — <i>жесткий диск</i> .
0x41 (65)	1	0	Зарезервировано для FAT32
0x42 (66)	1		Сигнатура расширенного загрузочного сектора.
0x43 (67)	4		Серийный номер тома. Случайное число, которое генерируется при форматировании
0x47 (71)	11		Метка тома
0x52 (82)	8		Тип файловой системы (12-, 16-, 32-разрядная)
0x5A (90)	420		<i>Начало кода системного загрузчика</i>
0x1FE (510)	2		Сигнатура (слово AA55)

Поскольку содержимое загрузочной записи имеет для ОС важное значение, то *обычно имеются резервные копии этой записи*.

Резервная загрузочная запись, как правило, располагается *в секторах 7-9 раздела*.

Таблица размещения файлов (FAT) – это *массив целых чисел* с длиной, равной количеству кластеров раздела файловой системы.

Номер элемента этого массива – это *номер кластера в разделе файловой системы*.

Отдельный элемент массива:

- для **FAT16** — это 2-х байтовые числа;
- для **FAT32** — это 4-х байтовые числа.

Значение элемента таблицы FAT — это *ссылка на следующий номер элемента таблицы FAT*. В таблице 1.2, представлены возможные значения таблицы FAT.

Таблица 1.2 - Значения элементов таблицы FAT32 (используется 28 бит)

Значение	Описание
0x00000000	Свободный кластер
0x00000002 – 0x0FFFFFFF	Номер следующего кластера в цепочке
0x0FFFFFFF0 – 0x0FFFFFFF6	Зарезервированный кластер
0x0FFFFFFF7	Плохой кластер
0x0FFFFFFF8 – 0x0FFFFFFF	Последний кластер в цепочке

Замечание

В файловой системе FAT, между копиями загрузочной записи раздела и корневым каталогом, *находятся две копии таблиц FAT*.

При загрузке ОС таблица **FAT** загружается в оперативную память компьютера. Для больших файловых систем таблица **FAT** может загружаться частями.

За таблицами **FAT** располагается *корневой каталог файловой системы (Root Directory)*, размером:

- **512 байт**, для **FAT16**.
- **2048 байт**, для **FAT32**, сейчас — до 65535 элементов записей по 32 байта.

Корневой каталог файловой системы – список записей по **32 байта**.

Исторически, сложилась ситуация, что файловая система FAT содержит *два типа имён*:

- *короткие имена* — до восьми байт имя и три байта расширение имени;
- *длинные имена* — до 255 символов в формате **UTF-16LE**.

Формат записи короткого имени файла вмещается в одну 32-байтовую запись.

Структура этого формата представлена в таблице 1.3.

Одиннадцатый байт записи содержит поле «**Атрибуты**», значения которого представлены в таблице 1.4.

Для длинных имён файлов, в формате **Unicode-16**, используется несколько 32-байтовых записей каталога в специальном формате. Один элемент, такого фор-

мата записи, показан на рисунке 1.5 и имеет несколько полей.

Таблица 1.3 - Основной вариант записи для короткого имени FAT32

Смещение			Описание
Hex	Dec	Длина поля	
00h	0	8 байт	Имя файла
08h	8	3 байт	Расширение файла
0Bh	11	1 байт	Атрибуты файла
0Ch	12	2 байт	Зарезервировано для NT
0Eh	14	2 байт	Время создания файла
10h	16	2 байт	Дата создания файла
12h	18	2 байт	Дата последнего доступа
14h	20	2 байт	Старшее слово номера начального кластера
16h	22	2 байт	Время последней записи
18h	24	2 байт	Дата последней записи
1Ah	26	2 байт	Младшее слово начального кластера
1Ch	28	4 байт	Размер файла в байтах

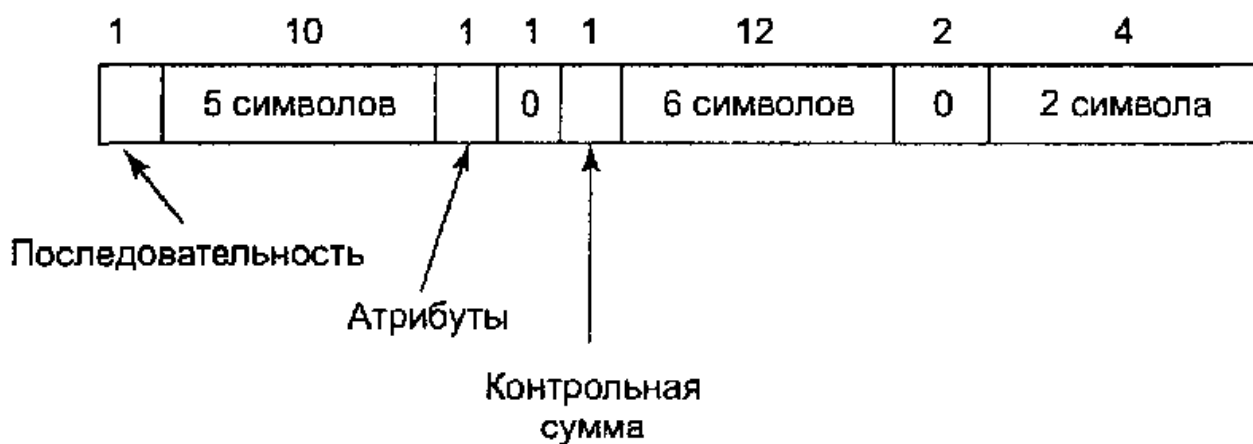


Рисунок 1.5 - Структура элемента записи для длинного имени файла

Таблица 1.4 - Значения 11-го байта атрибутов

7	6	5	4	3	2	1	0	Hex	Значение
0	0	0	0	0	0	0	1	01h	Только чтение
0	0	0	0	0	0	1	0	02h	Скрытый
0	0	0	0	0	1	0	0	04h	Системный
0	0	0	0	1	0	0	0	08h	Метка тома
0	0	0	1	0	0	0	0	10h	Подкаталог
0	0	1	0	0	0	0	0	20h	Архивный (измененный)
0	1	0	0	0	0	0	0	40h	Зарезервировано
1	0	0	0	0	0	0	0	80h	Зарезервировано

Область данных — все остальные кластеры раздела. Они используются для хранения *подкаталогов* и *файлов*.

Подкаталоги — это файлы, содержащие последовательности 32-битных записей *имён файлов и подкаталогов*, указанного выше формата.

Содержимое файлов — данные *некоторой последовательности кластеров*.

Последовательность кластеров — произвольная и неупорядоченная, но *отражена в виде цепочки*, в таблице FAT.

Замечание

Чем больше размер кластера, тем больший размер раздела может адресоваться и повышается скорость обработки файлов, но уменьшается эффективность их хранения.

1.1.4 Структура файловой системы EXT2FS

ext2 (ext2fs) - *Second Extended File System* — базовая файловая система ОС Linux.

Разработана Реми Кардом и представлена *в январе 1993 года*.

Поддерживает файлы размером — до *16 ГБайт - 2 ТБайт*.

Размер файловой системы — до *2 - 32 Тбайт*.

Максимум файлов — до 10^{18} штук.

ext2 является преемницей **ext** - *Extended File System*, представленной *в апреле 1992 года*.

Ext - поддерживала файлы размером *до 2 ГБайт* и длину имени файла до *255 байт*.

Структура дискового раздела ext2fs представляет *последовательность группы блоков*, которые нумеруются, начиная с 1.

Загрузочная запись – принадлежит 1-й группе блоков	Группа блоков 1	Группа блоков 2	...	Группа блоков n
---	--------------------	--------------------	-----	--------------------

Каждая группа блоков состоит из *последовательности блоков*, которые также нумеруются, начиная с 1.

Каждая группа блоков имеет *одинаковое число блоков*, кроме последней, которая может быть неполной.

Начало каждой группы блоков имеет *адрес*, который может быть получен как:

$$\text{Address} = (\text{номер_группы} - 1) * (\text{число_блоков_в_группе})$$

Размер блока может быть *1, 2 или 4 килобайта*, что определяется в момент форматирования – при создании файловой системы.

Блок является *адресуемой единицей дискового пространства*.

Каждая группа блоков имеет одинаковое строение:

Супер-блок	Описание группы блоков (Group Descriptors)	Битовая карта блоков (Block Bitmap)	Битовая карта индексных дескрипторов (Inode Bitmap)	Таблица индексных дескрипторов (Inode Table)	Область блоков данных
------------	--	-------------------------------------	---	--	-----------------------

Такая структура служит повышению производительности файловой системы:

- **сокращается расстояние** между таблицей индексных дескрипторов и блоками данных;
- **сокращается время поиска** нужного места головками в процессе операций записи/считывания файла.

Первый элемент группы блоков - **суперблок**, который — одинаков для всех групп и имеет размер **1024 байта**.

Все остальные элементы - индивидуальны для каждой группы.

Первые 1024 байта раздела занимает **загрузочная запись раздела**.

Суперблок, хранится в первой группе блоков и имеет **смещение 1024 байта**.

Наличие нескольких копий суперблока объясняется чрезвычайной важностью этого элемента файловой системы.

Дубликаты суперблока используются при восстановлении файловой системы после сбоев.

Информация, хранящаяся в суперблоке, используется для организации доступа к остальным данным на диске. В суперблоке определяется размер файловой системы, максимальное число файлов в разделе, объем свободного пространства и содержится информация о том, где искать незанятые участки.

При запуске ОС, суперблок считывается в память и все изменения файловой системы вначале находят отображение в копии суперблока, находящейся в ОЗУ, и записываются на диск только периодически.

При выключении системы, суперблок обязательно должен быть записан на диск.

Таблица 1.5 - Структура суперблока

Название поля	Тип	Комментарий
s_inodes_count	ULONG	Число индексных дескрипторов в файловой системе
s_blocks_count	ULONG	Число блоков в файловой системе (4 GBlocks)
s_r_blocks_count	ULONG	Число блоков, зарезервированных для суперпользователя
s_free_blocks_count	ULONG	Счётчик числа свободных блоков
s_free_inodes_count	ULONG	Счётчик числа свободных индексных дескрипторов
s_first_data_block	ULONG	Первый блок, который содержит данные. В зависимости от размера блока, это поле может быть равно 0 или 1.

s_log_block_size	ULONG	Индикатор размера логического блока: $0 = 1 \text{ КБ}$; $1 = 2 \text{ КБ}$; $2 = 4 \text{ КБ}$; $3 = 8 \text{ КБ}$.
s_log_frag_size	LONG	Индикатор размера фрагментов
s_blocks_per_group	ULONG	Число блоков в каждой группе блоков
s_frags_per_group	ULONG	Число фрагментов в каждой группе блоков
s_inodes_per_group	ULONG	Число индексных дескрипторов (inodes) в каждой группе блоков
s_mtime	ULONG	Время, когда в последний раз была смонтирована файловая система.
s_wtime	ULONG	Время, когда в последний раз производилась запись в файловую систему
s_mnt_count	USHORT	<i>Счётчик числа монтирований</i> файловой системы. Если этот счётчик достигает значения, указанного в следующем поле (s_max_mnt_count), файловая система должна быть проверена (это делается при перезапуске), а счётчик обнуляется.
s_max_mnt_count	SHORT	Число, определяющее, сколько раз может быть смонтирована файловая система
s_magic	USHORT	" <i>Магическое число</i> " ($0xEF53$), указывающее, что файловая система принадлежит к типу ex2fs
s_state	USHORT	Флаги, указывающее текущее состояние файловой системы: является ли она чистой (clean) и т.п.
s_errors	USHORT	Флаги, задающие процедуры обработки сообщений об ошибках (что делать, если найдены ошибки).
s_pad	USHORT	Заполнение
s_lastcheck	ULONG	Время последней проверки файловой системы
s_checkinterval	ULONG	Максимальный период времени между проверками файловой системы
s_creator_os	ULONG	Указание на тип ОС, в которой создана файловая система
s_rev_level	ULONG	Версия (revision level) файловой системы.
s_reserved	ULONG[235]	Заполнение до 1024 байт ($235 * 4 = 940$)

Вслед за суперблоком расположено **Описание группы блоков** (Group Descriptors).

Отдельный элемент этого описания имеет *структуру длиной 32 байта*.

Таблица 1.6 - Структура отдельного элемента описания группы блоков

Название поля	Тип	Назначение
bg_block_bitmap	ULONG	Адрес блока, содержащего битовую карту блоков (block bitmap) данной группы
bg_inode_bitmap	ULONG	Адрес блока, содержащего битовую карту индексных дескрипторов (inode bitmap) данной группы
bg_inode_table	ULONG	Адрес блока, содержащего таблицу индексных дескрипторов (inode table) данной группы
bg_free_blocks_count	USHORT	Счётчик числа свободных блоков в данной группе
bg_free_inodes_count	USHORT	Число свободных индексных дескрипторов в данной группе
bg_used_dirs_count	USHORT	Число индексных дескрипторов в данной группе, которые являются каталогами
bg_pad	USHORT	Заполнение
bg_reserved	ULONG[3]	Заполнение

Размер описания группы блоков можно вычислить как:

$$N = (\text{размер_группы_блоков_в_ext2} * \text{число_групп}) / \text{размер_блока}$$

Информация, которая хранится в описании группы блоков, используется, чтобы найти:

- битовую карту блоков;
- битовую карту индексных дескрипторов,
- таблицу индексных дескрипторов.

Битовая карта блоков (*block bitmap*) - это структура, каждый бит которой показывает, отведён ли соответствующий ему блок какому-либо файлу.

Если бит равен 1, то блок занят.

Эта карта также служит для поиска свободных блоков в тех случаях, когда надо выделить место под файл.

Битовая карта блоков занимает число блоков, равное:

$$\text{b_blocks} = (\text{число_блоков_в_группе} / 8) / \text{размер_блока}$$

Битовая карта индексных дескрипторов выполняет аналогичную функцию по отношению к таблице индексных дескрипторов: показывает *какие именно дескрипторы заняты и каким типом файлов*.

Таблица индексных дескрипторов:

После битовой карты индексных дескрипторов, следует *таблица индексных дескрипторов файлов* - *Inode Table*.

Каждая строка *Inode Table* имеет *размер 128 байт* и имеет *структуру индексного дескриптора файлов*, рассмотренного ниже.

Количество строк *Inode Table*, а следовательно, и размер таблицы, *задаётся при создании файловой системы*.

Замечание

Каждый файл требует одной записи в таблицу дескрипторов. Поэтому *размер этих таблиц нужно предусмотреть заранее*.

Среди индексных дескрипторов имеется *несколько значений дескрипторов*, которые *зарезервированы для специальных целей* и играют особую роль в файловой системе.

Замечание

Значения этих дескрипторов, которые перечислены в таблице 1.7, не могут быть использованы для других целей.

Таблица 1.7 - Специальные индексные дескрипторы

Идентификатор	Знач.	Описание
EXT2_BAD_INO	1	Индексный дескриптор, в котором перечислены <i>адреса дефектных блоков на диске</i> (Bad blocks inode)
EXT2_ROOT_INO	2	Индексный дескриптор <i>корневого каталога</i> файловой системы (Root inode)
EXT2_ACL_IDX_INO	3	ACL inode
EXT2_ACL_DATA_INO	4	ACL inode
EXT2_BOOT_LOADER_INO	5	Индексный дескриптор <i>загрузчика</i> (Boot loader inode)
EXT2_UNDEL_DIR_INO	6	Undelete directory inode
EXT2_FIRST_INO	11	<i>Первый незарезервированный</i> индексный дескриптор

Самый важный дескриптор в этом списке - *дескриптор корневого каталога*. *Каталог файловой системы* - файл, состоящий из записей переменной длины.

Таблица 1.8 - Структура файла корневого каталога

Название поля	Тип	Описание
inode	ULONG	Номер индексного дескриптора (индекс) файла
rec_len	USHORT	Длина этой записи

name_len	USHORT	Длина имени файла
name	CHAR[0]	Первый символ имени файла

Замечание

Отдельная запись в каталоге не может пересекать границу блока, то есть должна быть расположена целиком внутри одного блока.

Если очередная запись каталога не помещается целиком в данном блоке, то она переносится в следующий блок, а предыдущая запись продолжается таким образом, чтобы она заполнила блок до конца.

Индексные дескрипторы файлов:

Вся информация о файле хранится в индексном дескрипторе (**Inode**).

Число файлов, которое может быть создано в файловой системе, *ограничено числом индексных дескрипторов*, которое:

- *либо явно задаётся*, при создании файловой системы;
- *либо вычисляется*, исходя из физического объёма дискового раздела.

Размер индексного дескриптора = $32 * 4 = 128$ байт.

Структура индексного дескриптора приведена в таблице 1.9.

Таблица 1.9 - Структура индексного дескриптора файла (128 байт)

Название поля	Тип	Описание
i_mode	USHORT	Тип файла и права доступа к данному файлу.
i_uid	USHORT	Идентификатор владельца файла (Owner UID).
i_size	ULONG	Размер файла в байтах (4 байта).
i_atime	ULONG	Время последнего обращения к файлу (Access time).
i_ctime	ULONG	Время создания файла.
i_mtime	ULONG	Время последней модификации файла.
i_dtime	ULONG	Время удаления файла.
i_gid	USHORT	Идентификатор группы (GID).
i_links_count	USHORT	Счётчик числа связей (Links count) — (<i>число имён файла ?</i>)
i_blocks	ULONG	Число блоков, занимаемых файлом.
i_flags	ULONG	Флаги файла (File flags)
i_reserved1	ULONG	Зарезервировано для ОС
i_block	ULONG[15]	Указатели на блоки данных, которые рассмотрены далее в разделе «Система адресации данных»
i_version	ULONG	Версия файла (для NFS)
i_file_acl	ULONG	ACL файла - Access Control List – список доступа
i_dir_acl	ULONG	ACL каталога - Access Control List – список доступа

<code>i_faddr</code>	ULONG	Адрес фрагмента (Fragment address)
<code>i_frag</code>	UCHAR	Номер фрагмента (Fragment number)
<code>i_fsize</code>	UCHAR	Размер фрагмента (Fragment size)
<code>i_pad1</code>	USHORT	Заполнение
<code>i_reserved2</code>	ULONG[2]	Зарезервировано

Среди множества полей индексного дескриптора файла, более подробно мы рассмотрим ***i_mode*** и ***i_block***.

Поле ***i_mode*** - тип и права доступа к файлу.

Оно является *двухбайтовым словом*, каждый бит которого служит *флагом*, индицирующим *отношение файла к определённому типу* или *установку одного конкретного права на файл* (таблица 1.10).

Таблица 1.10 - Поле прав доступа к файлу

<i>Идентификатор</i>	<i>Значение</i>	<i>Назначение флага (поля)</i>
S_IFMT	F000	Маска для типа файла
S_IFSOCK	A000	Доменное гнездо (socket)
S_IFLNK	C000	Символическая ссылка
S_IFREG	8000	Обычный (regular) файл
S_IFBLK	6000	Блок ориентированное устройство
S_IFDIR	4000	Каталог
S_IFCHR	2000	Байт ориентированное (символьное) устройство
S_IFIFO	1000	Именованный канал (fifo)
S_ISUID	0800	SUID - бит смены владельца
S_ISGID	0400	SGID - бит смены группы
S_ISVTX	0200	Бит сохранения задачи (sticky bit)
S_IRWXU	01C0	Маска прав владельца файла
S_IRUSR	0100	Право на чтение
S_IWUSR	0080	Право на запись
S_IXUSR	0040	Право на выполнение
S_IRWXG	0038	Маска прав группы
S_IRGRP	0020	Право на чтение
S_IWGRP	0010	Право на запись

S_IXGRP	0008	Право на выполнение
S_IRWXO	0007	Маска прав остальных пользователей
S_IROTH	0004	Право на чтение
S_IWOTH	0002	Право на запись
S_IXOTH	0001	Право на выполнение

Система адресации данных — это одна из самых существенных составных частей файловой системы *ext2fs*. Именно она позволяет находить нужный файл среди множества как пустых, так и занятых блоков на диске.

В *ext2fs*, система адресации реализуется полем *i_block* индексного дескриптора файла. Поле *i_block* в индексном дескрипторе файла представляет собой *маску из 15 адресов блоков*, что наглядно демонстрируется рисунком 1.6.

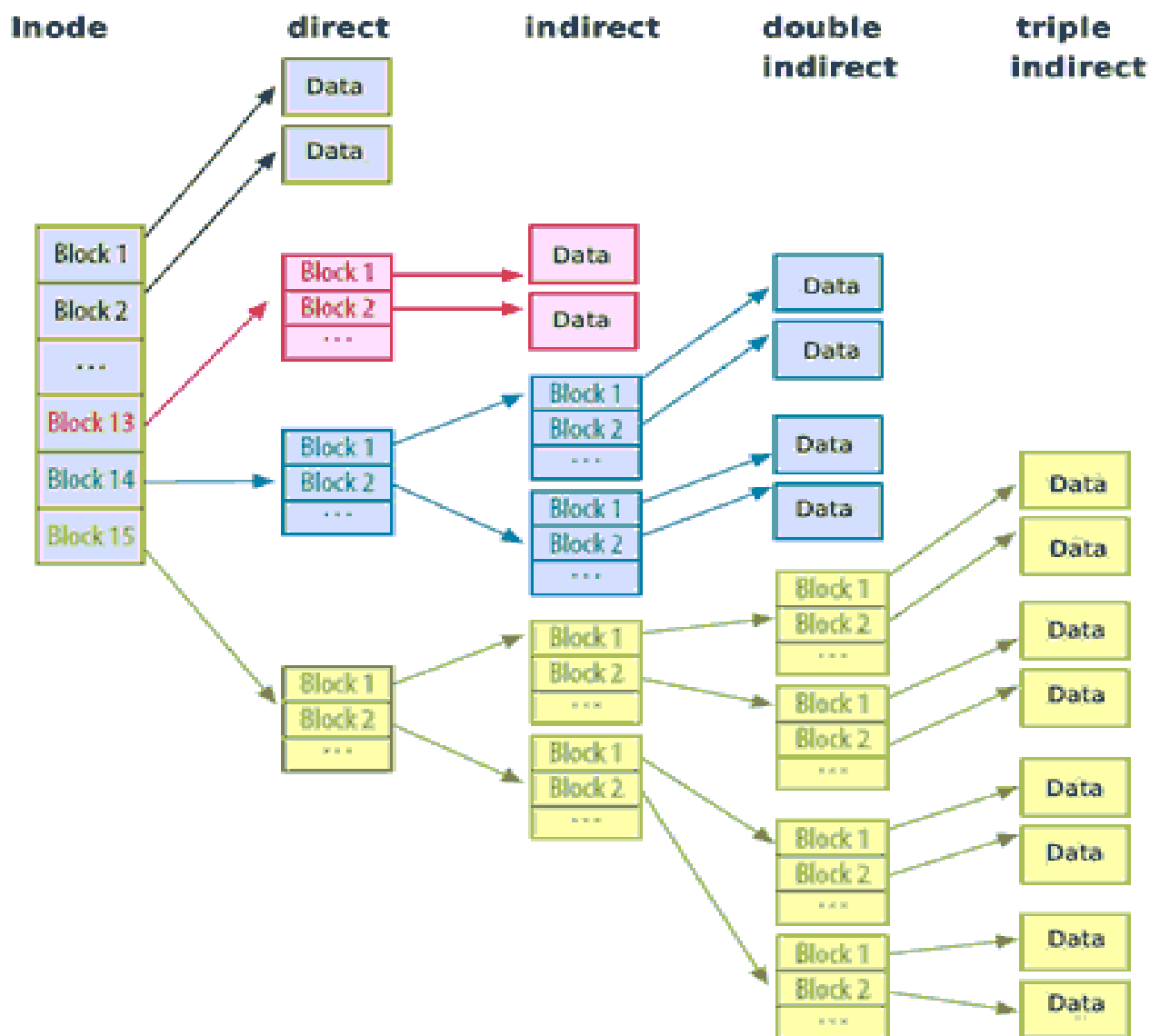


Рисунок 1.6 - Адресация блоков файла с помощью *i_block*

Первые 12 адресов в этом массиве `i_block[0 - 11] = EXT2_NDIR_BLOCKS [0 - 11]` - **прямые ссылки** (адреса) на номера блоков, в которых хранятся данные из файла.

Следующий адрес `i_block[12] = EXT2_IND_BLOCK` является **косвенной ссылкой на блок**, в котором хранится *список следующих адресов* блоков с данными для этого файла:

Количество адресов = (размер_блока / размер_ULONG)

Следующий адрес `i_block[13] = EXT2_DIND_BLOCK` указывает на **блок двойной косвенной адресации** (double indirect block). Этот блок содержит список адресов блоков, которые в свою очередь содержат списки адресов следующих блоков данных того файла, который задаётся данным индексным дескриптором:

Количество адресов = (размер_блока / размер_ULONG)²

Последний адрес `i_block[14] = EXT2_TIND_BLOCK` указывает на **блок тройной косвенной адресации**:

Количество адресов = (размер_блока / размер_ULONG)³

Все оставшееся место, в группе блоков, отводится для хранения файлов.

Таким образом, прочитав суперблок, мы можем:

- определить адреса групп блоков данных прочитать групповые дескрипторы;
- из групповых дескрипторов определяем положение и размер индексных дескрипторов файлов;
- просматривая таблицу индексных дескрипторов файлов, мы определяем, что описывает дескриптор: файл, каталог, символьное или блочное устройство, другие объекты;
- по дескрипторам каталогов мы читаем блоки данных, в которых записаны имена файлов, каталогов, других устройств и соответствующие номера записей в таблице индексных дескрипторов файлов;
- номера свободных блоков и индексных дескрипторов файлов мы определяем по соответствующим битовым таблицам (картам).

1.1.5 Сравнение файловых систем

Рассмотренные выше структуры файловых систем **FAT32** и **ext2fs** демонстрируют два разных подхода к обработке данных:

- **простота управления**, характерную для **FAT32**;
- **скорость обработки больших файлов**, характерную для **ext2fs**.

В любом случае обе файловые системы имеют общую метаструктуру, которая определяется наличием:

- **секторов** загрузчика ОС;
- **суперблока**, характеризующего файловую систему и раздел в котором она расположена;

- *набора таблиц* для разметки и учёта использованных блоков данных;
- *корневого каталога*, с которого начинается логическое построение ФС;
- *набора блоков данных*, в которых размещаются подкаталоги и файлы ФС.

Фактически, все известные файловые системы отличаются только *способом реализации этих элементов метаструктуры*.

Естественным образом, файловые системы *FAT32* и *ext2fs* имеют как свои преимущества, так и недостатки.

Преимущества FAT32:

- *простота реализации* и эффективность использования всего физического пространства блочных устройств малой ёмкости;
- *широкая известность* и распространённость, делающая ее «*универсальным хранилищем*» временных ФС.

Недостатки FAT32:

- практическая ограниченность размера ФС и слабая защищённость, делающая её непригодной для современных ОС и технологий хранения информации;
- высокая фрагментированность ФС в процессе эксплуатации, что снижает скорость её работы и негативно воздействует на устройства хранения данных.

Именно эти недостатки потребовали от корпорации Microsoft, **в марте 1993 года**, замены FAT32 на более сложную и современную ФС: **NTFS**.

Файловая система ext2 берет своё начало от ФС **minix**.

Файловая система **ext** - это **Extended minix**.

Преимущества ext2:

- *высокое быстроедействие*;
- *поддержка файлов до 2 ТБайт*.

Недостатки ext2:

- *отсутствие журналирования*, снижающее её надёжность;
- *недостаточные*, по современным меркам, *размеры* поддерживаемых разделов ФС и размера файлов.

Указанные недостатки ext2 потребовали дальнейшего ее усовершенствование:

- **ext3** — *журналируемая ФС*, которая появилась в ноябре 2001 года; поддерживает размер файлов до **16 ТБайт** и размер раздела до **32 ТБайт**;
- **ext4** — *журналируемая ФС*, которая появилась в октябре 2008 года; поддерживает размер файлов до **16 ТБайт** и размер раздела до **1 ЭБайт**.

Журналируемая файловая система — это ФС, которая сохраняет список изменений, проводимых с файловой системой, перед фактическим их осуществлением.

Записи будущих фактических изменений хранятся в отдельной части ФС, называемой журналом (*journal*) или логом (*log*). Как только изменения файловой системы внесены в журнал, она применяет эти изменения к файлам или метаданным, а затем удаляет эти записи из журнала.

Записи журнала организованы в наборы связанных изменений файловой системы.

Замечание

Журналируемые записи первоначально появились в управлении базами данных, для которых характерно целостное управление большими массивами данных. С помощью журналирования также реализуется *механизм транзакций*.

1.2 Стандартизация структуры ФС

Предыдущий учебный материал даёт описание блочного устройства на уровне следующих свойств ФС:

- *секторов, LBA и разделов* винчестера: подразделы 1.1 и 1.2;
- *специализированных блоков отдельных ФС*, размещаемых в отдельном разделе винчестера: подразделы 1.3 — 1.6;

Данный учебный материал рассматривает структуру ФС как организованный набор файлов и директорий, видимый программами (процессами) пользовательского режима работы отдельных ОС.

В общем случае, структура ФС на уровне файлов и каталогов может быть произвольной, за некоторыми ограничениями, например:

- *обязательно наличие корневой ФС* ОС;
- *обязательно наличие корневого каталога* отдельных ФС.

На рисунке 1.7 приведён пример дерева каталогов и файлов некоторой ФС типа UNIX.

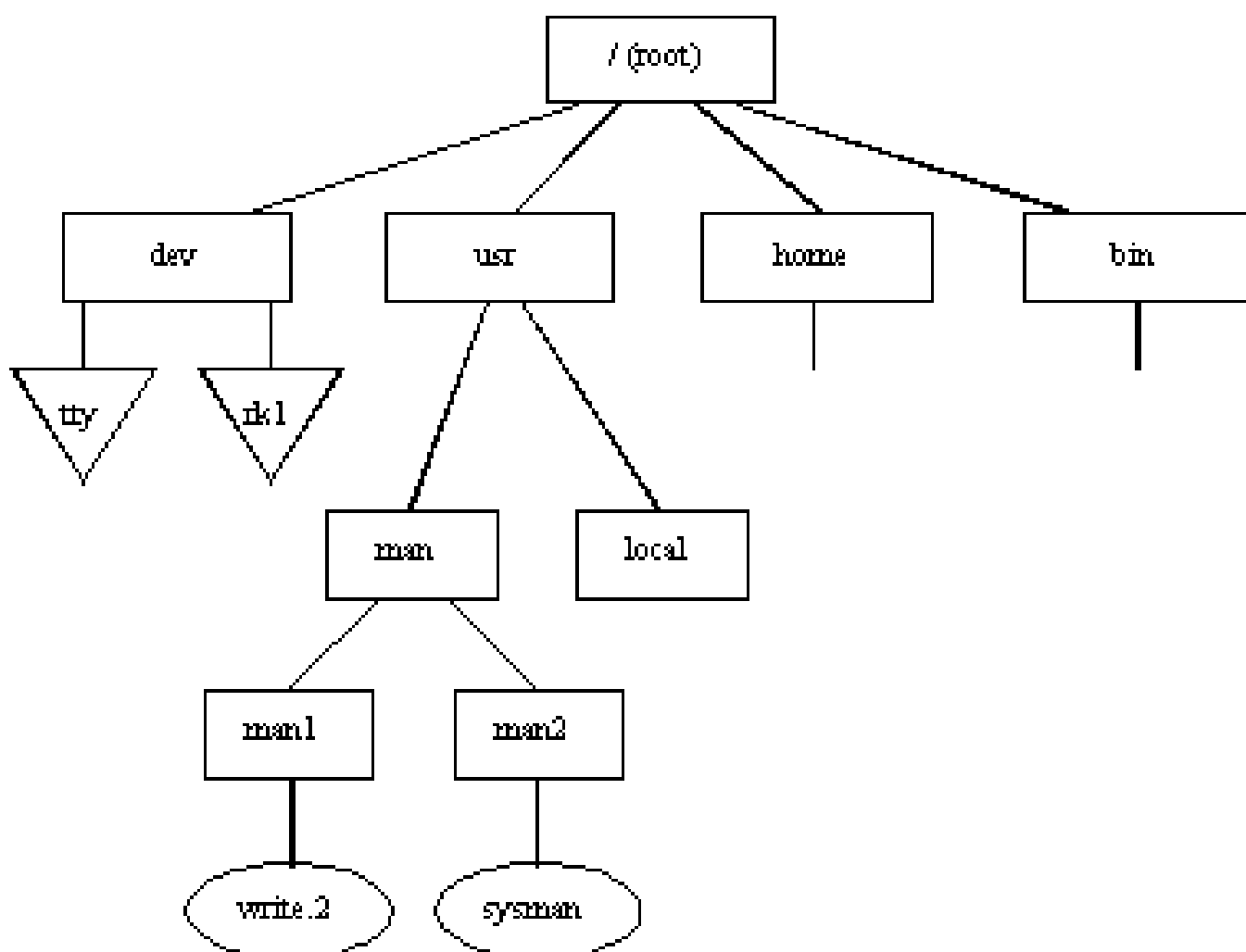


Рисунок 1.7 - Пример структуры ФС типа UNIX

Хорошо видно, что это дерево ФС логически структурировано в прикладном плане, например:

- *dev* — директория размещения узлов устройств ЭВМ (devices);
- *usr* — директория ориентированная на универсальные ресурсы ОС;
- *home* — директория для рабочих (домашних) областей пользователей;
- *bin* — директория расположения исполняемых файлов ОС (binary).

В августе 1993 года, в рамках проекта **GNU**, стал разрабатываться стандарт на структуру файловой системы (*Filesystem Standard*), который был выпущен в феврале 1994 года и получил обозначение **FSSTND**.

Позже, в 1996 году, была организована группа **Free Standard Group (FSG)**, к которой присоединились и другие разработчики ОС.

FSG продолжила разработку структуры иерархии ФС, которая была бы пригодна для всех UNIX подобных систем. Такой стандарт получил название **Filesystem Hierarchy Standard (FHS)**.

В общем случае, файловая система Linux разделена на *три крупных уровня иерархий*. Кратко рассмотрим каждый из этих уровней.

Первый уровень иерархии ФС составляют каталоги, которые управляются пользователем **root** и другими *администраторами системы*.

/	<i>Корневой каталог</i> . Должен содержать все, что необходимо для старта и запуска ОС, а также на случай восстановления её из резервной копии.
/bin	Необходимые программы (бинарные файлы).
/boot	Ядро ОС и файлы загрузчиков, например, LILO, GRUB, ...
/dev	Устройства, узлы. Все устройства доступны в виде файлов.
/etc	Файлы конфигурации.
/etc/X11	Пример, файлы конфигурации для X Window.
/home	Домашние каталоги пользователей.
/home/reznik	Пример, домашний каталог пользователя reznik.
/lib	Разделяемые библиотеки, необходимые для работы программ, находящиеся в каталогах <i>/bin</i> , <i>/sbin</i> и др.
/lib/modules	Пример, модули ядра ОС.
/media	Каталог монтирования временных ФС — таких, как floppy и другие.
/mnt	Точка для временного монтирования файловых систем.
/opt	Каталог для пакетов программ для опциональной установки.
/opt/Office50	Пример, пакет Star Office 5.0 (в данном случае).
/proc	<i>Специальные виртуальные файлы</i> , обеспечивающие доступ к функциям ядра UNIX или Linux.

/root	Домашний каталог пользователя root.
/sbin	Необходимые системные программы (бинарные файлы для root).
/tmp	<i>Временные файлы</i> для всех программ. Эти файлы должны уничтожаться системой при каждой её перезагрузке.
/usr	Второй уровень иерархии — (usr — <i>UNIX system resource</i>)
/var	Изменяемые данные (сохраняемые при перезагрузке ОС).

Второй уровень иерархии ФС, по замыслу, содержит *данные, которые доступны только для чтения* и могут использоваться всеми пользователями или несколькими компьютерами сети. В идеале изменяться могут только файлы каталога **/usr/local**, где администратор ОС может устанавливать новые программы, которые не поставляются с системой.

/usr/X11R6	Система X Window, версия 11.6
/usr/X386	Система X Window, версия 11.5
/usr/bin	Программы, предназначенные для пользователей.
/usr/games	Игры и развлечения.
/usr/include	Файлы заголовков для программ на языке C.
/usr/lib	Библиотеки.
/usr/local	Локальная иерархия (корень третьего уровня иерархии) . Обычно сюда монтируются всякие дополнения.
/usr/sbin	Системные программы.
/usr/share	Архитектурно независимые данные:
/usr/share/dict	Словари.
/usr/share/doc	Документация в свободных форматах.
/usr/share/games	Игры.
/usr/share/info	Документация в формате GNU info.
/usr/share/locale	Данные для системной локали.
/usr/share/man	Документация в формате man .
/usr/share/nls	Данные для поддержки национальных языков (NLS).
/usr/share/misc	Все остальные данные.
/usr/share/terminfo	База данных terminfo.
/usr/share/tmac	Макросы для системы troff.
/usr/share/zoneinfo	База данных и конфигурация местного времени .
/usr/src	Исходные тексты (ядро ОС).
/usr/src/linux	Исходные тексты ядра Linux.

Третий уровень иерархии ФС — *подобен второму* и продолжается в директории `/usr/local`.

Далее, более подробно рассмотрим корневую директорию изменяемых данных `/var`, поскольку она содержит множество системных данных ОС.

<code>/var</code>	<i>Каталог</i> , содержимое которого может меняться при работе системных программ.
<code>/var/cache</code>	Временно сохраняемые файлы различных программ (браузеров Internet, программы <i>man</i> , сервера шрифтов и другие).
<code>/var/games</code>	Изменяемые файлы для игр из <code>/usr/games</code> .
<code>/var/lock</code>	Lock-файлы, указывающие, что то или иное устройство (обычно модем) занято.
<code>/var/log</code>	Системные журнальные файлы (ссылка на <code>/run/lock</code>).
<code>/var/mail</code>	Почтовые ящики пользователей.
<code>/var/opt</code>	Изменяемые файлы для программ из <code>/opt</code> .
<code>/var/run</code>	Файлы, имеющие отношение к запущенным в настоящий момент программам, например, хранящие номера процессов (сейчас — ссылка на каталог <code>/run</code>).
<code>/var/spool</code>	<i>Данные, обработка которых отложена</i> (статьи из групп новостей, документы посланные на печать, письма, которые не удалось сразу послать из-за проблем со связью и другие).
<code>/var/state</code>	Данные, имеющие отношение к состоянию программ, например, backup -файлы, файлы текстовых редакторов, конфигурации, имеющие отношение ко всей системе и прочее.
<code>/var/tmp</code>	Временные файлы, которые <i>не должны</i> регулярно уничтожаться системой.
<code>/var/yp</code>	Файлы системы <i>NIS</i> (Network Information Services).

Замечание

- Видимое пользователям дерево ФС отличается от дерева ФС доступного ядру ОС:
- ядро ОС «видит» дерево виртуальной ФС (VFS);
 - пользователь «видит» только ветвь дерева, объявленного для пользователя как корень `root`;
 - современная тенденция — контейнерная технология, которая склоняется к минимизации видимой части ФС, повышая ее защищенность.

Управление видимостью дерева ФС осуществляется с помощью утилиты **chroot**. В частности, она используется во время выполнения процедуры **login**.

С другой стороны, видимая для пользователя ФС, создаёт для него *иерархию*

ческую систему координат, в пределах которой он принципиально может адресовать любой файл.

Следующий аспект работы пользователя состоит в том, что он всегда находится *в некоторой точке ФС*, которая связана с некоторой ее директорией:

- чтобы определить, где находится пользователь, следует использовать команду **pwd**, которая в качестве результата возвратит имя директории;
- чтобы перемещаться по ФС, пользователь должен использовать команду:

cd [директория],

где аргумент команды обрабатывается по следующим правилам:

- отсутствие аргумента *делает текущей директорию*, заданную системной переменной **HOME**;
- аргумент, начинающийся символом **/**, *задаёт абсолютные координаты директории*, начиная от корня ФС;
- аргумент, начинающийся любым допустимым символом, отличным от **/**, задаёт *относительные координаты директории*, начиная от текущей директории;
- *для уточнения* использования относительной адресации, можно использовать аргумент **./директория**;
- *для относительной адресации* от родительской директории, можно использовать аргумент **../директория**

Наиболее распространённые команды для работы с директориями:

- **mkdir директория**; - создание директории;
- **rmdir директория**; - удаление директории;
- **ls [опции] [шаблон]**; - вывод на консоль списка файлов и директорий;
- **cat файл**; - вывод содержимого файла на консоль.

Например,

```
$ ls -l
итого 864
drwxrwxrwx    5 root root  4096 окт.  6  2013 asu11upk3
drwxrwxrwx    3 root root  4096 июля  28  2013 asu12upk
-rw-rw-r--    1 vgr  vgr    18  дек.  22  2012 asu_tmp
drwxrwxr-x    2 vgr  vgr  4096 июня  1  02:32 bin
drwxrwxr-x    2 vgr  vgr  4096 дек.  26  2012 bin.old
-rw-r--r--    1 vgr  vgr  5860 янв.  5  2013 casper.log
-rw-rw-r--    1 vgr  vgr   180 сент. 18  2012 demo2.txt
-rw-rw-r--    1 vgr  vgr   490 сент. 18  2012 demo.txt
...
$
```

Кроме специального назначения директории **/dev**, о которой говорилось ранее, ФС имеет директорию **/proc**, в которую монтирована файловая система **proc**. В ней содержатся множество файлов, содержащих информацию о параметрах или

функциональных возможностях ядра ОС.

Например, чтобы узнать, какие типы ФС поддерживает ядро ОС, можно выполнить:

```
$ cat /proc/filesystems
nodev      sysfs
nodev      rootfs
nodev      bdev
nodev      proc
nodev      cgroup
nodev      cpuset
nodev      tmpfs
nodev      devtmpfs
nodev      debugfs
nodev      securityfs
nodev      sockfs
nodev      pipefs
nodev      anon_inodefs
nodev      devpts
nodev      ext3
nodev      ext4
nodev      ramfs
nodev      hugetlbfs
nodev      vfat
nodev      ecryptfs
nodev      fuseblk
nodev      fuse
nodev      fusectl
nodev      pstore
nodev      mqueue
$
```

Данный пример хорошо показывает, что *не все типы ФС* подразумевают наличие «физических» (**блочных**) устройств.

Другой пример, - файл **/proc/cmdline**, который содержит список параметров, переданных ядру во время загрузки ОС:

```
$ cat /proc/cmdline
BOOT_IMAGE=/boot/vmlinuz-3.5.0-21-generic root=/dev/sda5 ro
$
```

Замечание

Последний пример - чисто демонстрационный. Студенту самостоятельно следует определить и описать содержимое файла [/proc/cmdline](#).

1.2.1 Модули и драйверы ОС

Ядро ОС — это большая программа, работающая в собственном *защищённом (привилегированном) режиме*.

Любой процесс работает в *режиме пользователя*. Потому, *любой процесс имеет доступ к устройствам только через ядро ОС*.

Чтобы работать с конкретным устройством, ядро ОС имеет специальное ПО, которое называется **драйвером**.

ПО драйвера имеет два интерфейса:

- *интерфейс вызова драйвера ядром ОС*, который использует концепцию узла ФС или *node*, подразделяя устройства на *символьные* и *блочные*;
- *интерфейс устройства*, который использует сам драйвер для работы с конкретным устройством *через общую шину ЭВМ*.

Принципиальная разница между этими интерфейсами:

- *интерфейс вызова драйвера* ядром ОС *определяется разработчиками ОС* и различается в зависимости от её типа: разный интерфейс для разных ОС;
- *интерфейс устройства* определяется *архитектурой аппаратной части ЭВМ* и *конструктивными особенностями* конкретного устройства.

Учитывая большое разнообразие архитектур аппаратной части ЭВМ, *драйверы традиционно пишутся в виде модулей*, которые хранятся в ФС ОС и *загружаются по мере необходимости*.

Поскольку без некоторой части драйверов ОС не сможет даже загрузиться, такие драйверы включаются в ядро ОС **статически**. Обычно, перед компиляцией ядра ОС, запускается **программа-конфигуратор**, с помощью которой определяется: какие драйверы будут включаться в ядро **статически**, а какие будут подгружаться **динамически**: во время загрузки ОС или по мере необходимости.

Идея использования модулей ядра — очень привлекательна:

- *для разработки любого ПО* используются средства разработки, которые сами являются ПО и могут функционировать *только в среде уже загруженной ОС*;
- *статически скомпилированный драйвер* может вступить в конфликт с архитектурой аппаратной части ЭВМ, что может сделать ядро ОС *неработоспособным*;
- *стремление уменьшить размер ядра ОС*, которое является актуальным для *встроенных (embedded) систем* или в *проектах микроядерной архитектуры ОС*.

Для создания *модулей в ОС Linux* разработана специальная технология, которая широко освещена в соответствующей литературе. Например, корпорация IBM на сайте http://www.ibm.com/developerworks/ru/library/l-linux_kernel_01/ выпустила более 70-ти статей, посвящённых этой тематике.

В общем случае, **модуль** — **системное ПО ОС**, которое не обязательно является **драйвером**. Главное, что это ПО предназначено для работы в **защищённом режиме пространства ядра ОС**.

Использование модулей предполагает учёт их особенностей:

- **управление модулями** из режима пользователя осуществляется утилитами **insmod**, **rmmod**, **lsmod**, **modinfo** и **modprobe**;
- **разработка модулей** предполагает знание не только технологии их написания и отладки, но также - **архитектуры и ПО ядра ОС**;
- **вывод информации в пространство пользователя** модуль осуществляет посредством функции **printk(...)**, которая записывает сообщение в файл **/var/log/messages**;
- **для просмотра сообщений модулей ядра** используется утилита **dmesg**;
- **файлы конфигурации и сами модули** находятся в директориях ФС: **/etc** и **/lib/modules**.

1.2.2 Системные вызовы ОС по управлению устройствами и ФС

Один из общепринятых способов повышения мобильности ПО — обеспечение стандартизации окружения приложений. Это подразумевает стандартизацию **ПО ОС**, **утилит** и **программных интерфейсов** самих приложений. Таким средством является стандарт **POSIX** (*Portable Operating System Interface*), который описывает различные интерфейсы операционной системы.

Название POSIX предложено известным специалистом, основателем Фонда свободного программного обеспечения, Ричардом Столмэном. Наиболее современная версия стандарта POSIX, **в редакции 2003 г.**, основана на Техническом стандарте **Open Group IEEE Std 1003.1** и на международном стандарте **ISO/IEC 9945**.

По состоянию на 2001 год, стандарт содержал следующие **четыре части**:

1. **Основные определения** (термины, концепции и интерфейсы, общие для всех частей);
2. **Описание прикладного программного С интерфейса** к системным сервисам;
3. **Описание интерфейса** к системным сервисам на уровне командного языка и служебных программ;
4. **Детальное разъяснение** положений стандарта, обоснование принятых решений.

В дальнейшем, накапливались и были внесены многие мелкие исправления, учтённые **в редакции 2003 года**.

Стандарт POSIX — **обязательный элемент современной дисциплины разработки прикладных систем**.

Таким образом, рассмотрев основные понятия и требования стандарта POSIX, мы можем перейти к непосредственному обсуждению системных вызовов ОС. На рисунке 1.9, приведён список системных вызовов ОС, разделённых на группы:

- управление процессами;
- управление файлами;

- управление каталогами и файловыми системами;
- разные, которые нельзя полностью отнести к предыдущим группам.

Непосредственно к нашей теме, относятся группы *управление файлами*, а также *управление каталогами и файловыми системами*.

Примеры предыдущего раздела показывают, что зная структуру необходимых данных, можно одни функции реализовать через другие, «более мелкие», но общая парадигма системного ПО требует отношение к ним как к функциям одного уровня.

Замечание

Непосредственное применение системных вызовов предполагает изучение правил их вызова с использованием утилиты *man*, которая, собственно говоря, и была создана для этих целей. Наряду с низкоуровневыми системными вызовами, для работы с файлами и файловыми системами, используются более высокоуровневые функции, которые обеспечивают буферизованный ввод/вывод и подробно изложены в любом учебнике по языку C.

Вызов	Описание
<i>Управление процессами</i>	
<i>Pid=fork()</i>	Создает дочерний процесс, идентичный родительскому
<i>Pid=waitpid(pid, &statoc, options)</i>	Ожидает завершения дочернего процесса
<i>s=execve(name, argv, environp)</i>	Перемещает образ памяти процесса
<i>Exit(status)</i>	Завершает выполнение процесса и возвращает статус
<i>Управление файлами</i>	
<i>fd=open(file, how, ...)</i>	Открывает файл для чтения, записи
<i>s=close(fd)</i>	Закрывает открытый файл
<i>n=read(fd, buffer, nbytes)</i>	Читает данные из файла в буфер
<i>n=write(fd, buffer, nbytes)</i>	Пишет данные из буфера в файл
<i>Position=lseek(fd, offset, whence)</i>	Передвигает указатель файла
<i>s=stat(name, &buf)</i>	Получает информацию о состоянии файла
<i>Управление каталогами и файловой системой</i>	
<i>s=mkdir(name, mode)</i>	Создает новый каталог
<i>s=rmdir(name)</i>	Удаляет пустой каталог
<i>s=link(name1, name2)</i>	Создает новый элемент с именем
<i>s=unlink(name)</i>	Удаляет элемент каталога
<i>s=mount(special, name, flag)</i>	Монтирует файловую систему
<i>s=umount(special)</i>	Демонтирует файловую систему
<i>Разные</i>	
<i>s=chdir(dimame)</i>	Изменяет рабочий каталог
<i>s=chmod(name, mode)</i>	Изменяет биты защиты файла
<i>s=kill(pid, signal)</i>	Посылает сигнал процессу
<i>Seconds=time(&seconds)</i>	Получает время, прошедшее с 1 января 1970 г.

Рисунок 1.9 - Системные вызовы стандарта POSIX

1.2.3 Три концепции работы с блочными устройствами

Для того, чтобы уже созданные ФС стали доступны приложениям ОС, *должны быть созданы узлы (node)*, которые расположены в директории `/dev`.

Узлы ОС могут быть созданы из пространства пользователя:

- *с помощью системного вызова **mknod(...)*** из программы на языке **C**;
- *с помощью утилиты **mknod*** посредством команды языка **shell**.

Хотя любой из указанных способов предполагает указание всего четырёх параметров: *имя_узла*, *тип_узла*, *старший_номер_узла* и *младший_номер_узла*, такая ситуация создаёт множество проблем:

- *какое имя дать узлу*, чтобы оно мнемонически отображало *тип устройства* и было не очень длинным, для удобства работы с ним;
- *какой старший номер* присвоить устройству, чтобы он соответствовал типу устройства и *был стандартным для этого типа устройств*;
- *сколько должно быть младших номеров устройства*, чтобы удовлетворить все варианты использования ОС;
- *что делать с узлами устройств*, которых нет в архитектуре ЭВМ;
- *как узнать момент*, когда к ЭВМ устройство подключается динамически (*временное подключение и отключение устройств*).

Исторически, сложилось *три концепции (технологии)* именования и отслеживания актуальности наличия устройств: **manual**, **udev** и **systemd**.

Концепция manual (*ручная*) появилась первой. Она предполагает использование, упомянутых выше, системной функции **mknod(...)** или утилиты **mknod**.

Обычно, созданием устройств занимается *программа инсталлятор ОС*, которая проверяет архитектуру компьютера на наличие имеющихся устройств, а также создаёт узлы устройств: автоматически или по требованию администратора ОС.

Первоначально, именование и параметры узлов придумывались разработчиками ОС и были достаточно разнообразны. Со временем, стала проводиться стандартизация, учитывающая известные типы оборудования. Например:

- *почти все ОС* имеют устройство консоли `/dev/console`;
- *винчестеры, с контроллерами ATA*, стали именоваться `/dev/hda`, `/dev/hdb` и далее, а разделы, на которых фактически и создаются ФС - `/dev/hda1`, `/dev/hda2`, ..., `/dev/hdb1`, `/dev/hdb2`, ...
- *разделы винчестеров, с контроллерам SATA или SCSI*, стали именоваться как: `/dev/sda1`, `/dev/sda2`, ..., `/dev/sdb1`, `/dev/sdb2`, ...

Окончательно, процесс стандартизации завершился созданием скрипта shell — **MAKEDEV**, который упорядочил использование имён и других параметров узлов. Часто, этот скрипт можно было найти в директории `/dev` или `/sbin`. Сейчас эта технология ушла историю и больше не используется.

Замечание

Используя скрипт **MAKEDEV**, можно легко создать узлы с корректными именами и параметрами, но невозможно решить проблемы эффективного их использования. Особенно остро этот вопрос встал, когда корпорация Microsoft стала интенсивно призывать производителей оборудования внедрять *технология Plug and Play*.

Принципиальный аспект этой проблемы состоит в том, что драйвера устройств работают в пространстве ядра ОС. И хотя они могут определить факт подключения к ЭВМ нового устройства, *они не предназначены для посылки сигналов процессам*, работающим в режиме пользователя.

С начала 2000-х годов, в технологиях ОС, стала применяться файловая система *devfs*.

Цель этого проекта - решить основную проблему ОС UNIX/Linux, связанную как с катастрофически возрастающим объёмом узлов устройств в директории */dev*, так и с поддержкой технологии *PnP*.

Данный проект предполагал использование специального демона **devfsd**, который периодически обращался к ядру ОС с целью контроля за состоянием имеющегося оборудования:

- *при обнаружении подключения* нового оборудования, этот демон создавал в директории */dev* соответствующий узел (inode);
- *в случае обнаружения ситуации*, что оборудование отключилось, демон соответственно удалял узел из директории */dev*.

Существенным недостатком *devfs* является *неэффективное расходование ресурсов ЭВМ*, связанное с его постоянным циклическим запуском, поэтому современные системы **ОС UNIX/Linux** перешли на более новую технологию, известную как *udev*, используя для директории */dev* файловую систему **devtmpfs**.

Концепция **udev**, была реализована *в ноябре 2003 года*. Она решила проблему внедрения *технологии Plug and Play*.

Замечание

Здесь также следует отметить, что широкое внедрение графических оболочек ОС потребовало организовать *эффективный обмен сообщениями между демонами (фоновыми процессами) рабочего стола*.

Для этого была разработана *технология D-Bus* - специальная система межпроцессного взаимодействия, которая предоставляет *несколько шин (логических шин)*:

- *системную шину*, которая создаётся *при старте демона D-Bus* и обеспечивает взаимодействие демонов системных процессов;
- *сессионную шину*, которая создаётся для каждого пользователя, авторизовавшегося в системе, и обеспечивает взаимодействие его личных процессов.

Таким образом, при старте ОС *запускается демон udevd*, который, используя шину D-Bus, *отслеживает подключение новых устройств и создаёт для них нужные узлы*.

Когда устройства отключается от ЭВМ, **udev** удаляет соответствующие им

узлы, кроме тех, которые были созданы в ручную.

Одновременно, также решается *проблема поддержания в активном состоянии неиспользованных узлов*.

С апреля 2010 года, применительно к ОС Linux, стала внедряться «*Концепция systemd*». Ее цель — *унификация* взаимодействия между процессами режима пользователя.

В частности:

- *отслеживание динамики* подключения устройств (вместо *udev*);
- *эффективное отслеживание групп процессов*, порождённых системными вызовами *fork(...)* и объединённых одной прикладной целью (*cgroups*).

Замечание

ОС УПК АСУ, для работы с узлами устройств, использует эту современную концепцию *systemd*.

1.3 Разделы дисков и работа с ними

Когда созданы все ФС, на всех разделах блочных устройств, работа с файловой системой ОС становится достаточно простой:

- *необходимо определить точки монтирования* (директории) корневой ФС ОС для подключения к ним (монтирования) нужных сопутствующих ФС и затем, правильно записать желаемую конфигурацию в файле `/etc/fstab`;
- *выполнить команду `mount -a`*, и желаемая структура ФС ОС будет создана для последующей эксплуатации.

Хотя такой подход требует определённой квалификации пользователя и ориентирован больше на администраторов серверов, он вполне приемлем для многих случаев, когда аппаратная конфигурация ЭВМ является стабильной и не требует серьёзных обновлений.

Ситуация обостряется и становится критичной, когда необходимо провести модернизацию связанную с:

- *динамическим подключением других* («не родных») ФС;
- *подключением к сетевым ресурсам данных*, которые кроме проблем самой сети вынуждают использовать ФС, построенные на иных концепциях, например, ФС типа *FAT32* и *ntfs*, которые не используют атрибутов пользователя, группы и других.

В общем случае, эти проблемы решаются разработкой и использованием необходимых модулей ядра ОС, что уже было рассмотрено в подразделе 1.8. Но поскольку структура ФС достаточно сложна, а модули подключаются к ядру ОС, то:

- *размер ядра ОС* значительно увеличивается;
- *время разработки и последующей отладки ПО* модулей становится неприемлемо большим.

В 1985 году, компания Sun Microsystems предложила для ОС SunOS: **VFS** — *Virtual File System* (File System Switch) - *Виртуальную Файловую Систему*.

Виртуальная файловая система VFS:

- *Суть идеи* — расположить VFS между *приложениями* и *конкретными файловыми системами на внешних носителях*.
- Это позволяет *сконцентрировать в одном месте* пользовательский интерфейс для разных типов ФС, что *создаёт унифицированный эталон для разработки драйверов ОС*.

Таким образом, VFS обеспечивает *унифицированный программный интерфейс* к услугам файловой системы, причём безотносительно к тому, какой тип файловой системы (*vfat*, *ext2fs*, *nfs* и другие) имеется на конкретном физическом носителе.

В результате, VFS:

- *упрощает процесс создания драйверов ФС*, что повышает их надёжность;
- *сокращает время разработки*, что создаёт конкурентные преимущества самой ОС;
- *уменьшает время отладки и исправления ошибок*, что повышает актуаль-

ность их применения.

В любом случае, возможность применения тех или иных типов ФС зависит как от потребностей самих вычислительных систем, реализованных на ПО ОС, так и от возможности правильного управления имеющимися ФС.

Все равно, основу управления ФС составляют утилиты **mount** и **umount**, которые мы рассмотрим в следующем подразделе.

1.3.1 Монтирование и демонтирование устройств

Монтирование файловой системы — *подключение конкретного раздела* внешнего блочного устройства *к конкретному каталогу ФС*, в видимой пользователю общей части дерева корневой ФС.

Демонтирование файловой системы — *отключение конкретного раздела* внешнего блочного устройства от *конкретного каталога ФС*, в видимой пользователю общей части дерева корневой ФС.

Общее правило монтирования/демонтирования ФС:

- *монтирование осуществляется командой **mount** с указанием узла ФС и директории монтирования, при этом: **старое содержимое директории становится невидимым, а монтируемая ФС — видимой***;
- *демонтирование осуществляется командой **umount** с указанием или узла ФС или директории монтирования, при этом: **старое содержимое директории становится видимым, а демонтируемая ФС — невидимой***;

Простейший общий вид команды монтирования:

```
mount -t Тип Устройство Директория -о Опции;
```

Например, *монтирование устройства **/dev/sda2**, типа **ext2** в каталог **/mnt***, выглядит так:

```
$ mount -t ext2 /dev/sda2 /mnt
$
```

Команда **mount**, *без параметров*, выдаёт список монтированных в данный момент устройств:

```
/dev/sda5    on /          type ext4 (rw,errors=remount-ro)
proc        on /proc      type proc (rw,noexec,nosuid,nodev)
sysfs       on /sys       type sysfs (rw,noexec,nosuid,nodev)
none        on /sys/fs/fuse/connections type fusectl (rw)
...
/dev/sda2    on /mnt       type ext2 (rw)
```

Команда **umount**, *для указанного примера*, может быть выполнена в двух вариантах:


```
umount /dev/sda2;  
umount /mnt;
```

Замечание

Информация о действиях команды `mount` оперативно заносится в файл `/etc/mtab` в формате:

Устройство	он	Каталог	type	Тип	(Опции)
------------	----	---------	------	-----	---------

Для централизованного управления *монтированием* ФС под контролем ОС, используется файл `/etc/fstab`. Структура этого файла имеет вид:

Устройство	Каталог	Тип	Опции	fs_freq	fs_passno
------------	---------	-----	-------	---------	-----------

где

fs_freq — определяет, должна ли создаваться резервная копия этого раздела с помощью команды **dump**; значение **0** отменяет **dump**;

fs_passno — указывает в какую очередь данная ФС проверяется на целостность, при запуске Linux: 0 — проверка не требуется; 1 — для корневой ФС; 2 — для других ФС.

Демонстрационный пример `/etc/fstab`:

```
# /etc/fstab: static file system information.  
#  
# Use 'blkid' to print the universally unique identifier for a  
# device; this may be used with UUID= as a more robust way to name devices  
# that works even if disks are added and removed. See fstab(5).  
#  
# <file system> <mount point> <type> <options> <dump> <pass>  
# / was on /dev/sda2 during installation  
/dev/sda4 / ext2 errors=remount-ro 0 1  
# swap was on /dev/sda6 during installation  
/dev/sda6 none swap sw 0 0  
/dev/fd0 /media/floppy0 auto rw,user,noauto,exec,utf8 0 0
```

Замечание

Исторически, **опции**, с которыми монтируются ФС, создавались разработчиками разных ФС. Такая ситуация требует отдельного изучения опций для каждой ФС.

Кроме того, `/etc/fstab` имеет свои специфические опции.

Набор наиболее распространённых опций:

async	Весь ввод/вывод осуществляется асинхронно.
sync	Весь ввод/вывод осуществляется синхронно.
atime	Обновлять времена обращения (по умолчанию).
noatime	Времена обращения не обновляются.

<code>auto</code>	Система <i>может</i> быть смонтирована с опцией -a (для всех).
<code>noauto</code>	Система <i>не может</i> быть смонтирована с опцией -a (для всех).
<code>defaults</code>	Аналог <i>rw,suid,dev,exec,auto,nouser,async</i> .
<code>dev</code>	Файлы устройств ФС интерпретируются как устройства.
<code>nodev</code>	Файлы устройств запрещены.
<code>exec</code>	Право на запуск исполняемых файлов.
<code>noexec</code>	Запрет на запуск исполняемых файлов.
<code>suid</code>	Биты SUID и SGID <i>действуют</i> .
<code>nosuid</code>	Биты SUID и SGID <i>не действуют</i> .
<code>user</code>	Обычный пользователь может смонтировать систему. Подразумевает опции: <i>noexec,nosuid,nodev</i> .
<code>nouser</code>	Только root может монтировать систему.
<code>remount</code>	Перемонтировать уже смонтированную систему (например, для изменения опций системы).
<code>ro</code>	Смонтировать ФС только для чтения.
<code>rw</code>	Смонтировать ФС для чтения и записи.

1.3.2 Файловые системы `loopback`, `squashfs`, `overlayfs` и `fuse`

Наряду с рассмотренными выше ФС на разделах дисков и в ядре ОС, используется множество специальных ФС. Мы рассмотрим четыре из них: *loopback*, *squashfs*, *overlayfs* и *fuse*.

Термин *loopback* означает - «обратная петля». Применительно к нашей тематике он будет означать:

- *циклическое устройство*, когда мы будем говорить о блочном устройстве;
- *циклическая ФС*, когда мы будем говорить о файловой системе, которая создана в отдельном файле.

Узлы циклических устройств находятся в директории `/dev`. Следующий пример показывает вариант создания десяти циклических устройств: `/dev/loop1`, ..., `/dev/loop9`.

```
for x in 0 1 2 3 4 5 6 7 8 9; do
    [ -e /dev/loop$x ] || mknod /dev/loop$x b 7 $x
done
```

Циклические ФС обычно создаются в файлах, которые могут быть *сжатыми* и *зашифрованными*. Для примера рассмотрим создание ФС типа `ext2fs` в файле `/tmp/ramdisk` и последующее монтирование созданной ФС в директорию `/mnt/initrd`:

```
# Используемые константы
BLKSIZE=1024 # Размер блока в байтах
RDSIZE=4000  # Количество блоков в ФС
RBLOCK=0     # Число резервных блоков

# Создаем файл, забитый нулями
# (dd — преобразование и копирование файлов)
dd if=/dev/zero of=/tmp/ramdisk bs=$BLKSIZE count=$RDSIZE

# Создаём ФС в файле ext2fs
/sbin/mke2fs -F -m $RBLOCK -b $BLKSIZE /tmp/ramdisk $RDSIZE

# Монтируем ФС в файле к директории через устройство /dev/loop0
mount /tmp/ramdisk /mnt/initrd -t ext2 -o loop=/dev/loop0
```

Широкое распространение получила технология хранения файловых систем в *сильно сжатых файлах* для *live дистрибутивов*. Такой подход позволяет практически на прямую использовать технологии созданные для работы с ФС на CD дисках. Например, если устройство */dev/sda1* содержит файловую систему *ntfs*, в которой имеется файл */asu64upk/upkasu/usrfs.sfs*, то монтирование такой ФС (типа *squashfs*) к директории */usr* осуществляется двумя командами:

```
# Монтируем устройство /dev/sda1 в директорию /run/basefs
mount /dev/sda1 /run/basefs -t ntfs -o rw

# Монтируем ФС в файле .../usrfs.sfs к директории /usr
# через устройство /dev/loop1
mount /run/basefs/asu64upk/upkasu/usrfs.sfs /usr \
    -t squashfs -o loop=/dev/loop1
```

Другая специальная ФС - overlayfs — является вариантом реализации *каскадной файловой системы* для ОС *Ubuntu* и ее клонов.

Каскадная файловая система (КФС) — это виртуальная ФС, позволяющая «прозрачно видеть» две изолированные ФС как одну.

Работы над КФС ведутся с *середины 90-х годов*. В *январе 2007 года*, в некоторые дистрибутивы Linux, была включена КФС, названная *UnionFS*. Альтернативная разработка *Aufs (AnotherUnionFS)* ведётся с *2006 года*.

Каскадно-объединенное монтирование — *одновременное монтирование разных нескольких файловых систем как одну*, например для объединения ФС нескольких сайтов.

Overlayfs обеспечивает надёжное монтирование двух файловых систем на разных уровнях. Рассмотрим подробнее этот вопрос на примере:

- пусть устройство CDRом монтировано в директорию */cdrom* с возможностью *только чтение*;
- пусть в директорию */mnt/initrd* монтирован, созданный ранее *ramdisk*, который имеет возможность *чтения и записи*;

- проведём каскадно-объединённое монтирование в директорию `/tmp/mnt`, предполагая, что **CDROM** будет монтирован на нижнем уровне, а **ramdisk** — на верхнем;
- учтём, что в ядре ОС имеется имя **overlayfs**, которое не является устройством, но позволяет монтировать ФС.

Тогда, команда монтирования будет иметь вид:

```
# Команда каскадно-объединённого монтирования
mount -t overlay overlay /tmp/mnt \
      -o rw,upperdir=/mnt/initrd,lowerdir=/cdrom,workdir=/mnt/work
```

В результате такого действия, в директории `/tmp/mnt`, мы будем видеть все файлы директории `/cdrom`.

Кроме того, мы можем редактировать и использовать любой файл директории `/cdrom`, но сохраняться эти файлы будут в памяти **ramdisk**.

Последняя специальная ФС, которую мы кратко рассмотрим — **FUSE**.

FUSE — *Filesystem in Userspace* — файловая система в пространстве пользователя.

FUSE реализована в виде модуля для UNIX подобных ОС. Этот модуль *позволяет пользователям без привилегий создавать свои ФС*, без необходимости переписывать код ядра ОС. Это достигается за счёт запуска кода файловой системы в пространстве пользователя, в то время как *модуль FUSE предоставляет мост* для актуальных интерфейсов ядра.

На рисунке 1.8 показана схема взаимодействия системных вызовов пользователей с ядром ОС и ПО в пространстве пользователя. Хорошо видно, что:

- *схема взаимодействия* системного и прикладного ПО напоминает аналогичное взаимодействие *микроядра* и *серверных сервисов*;
- *кроме системной библиотеки glibc* необходима библиотека **libfuse**.

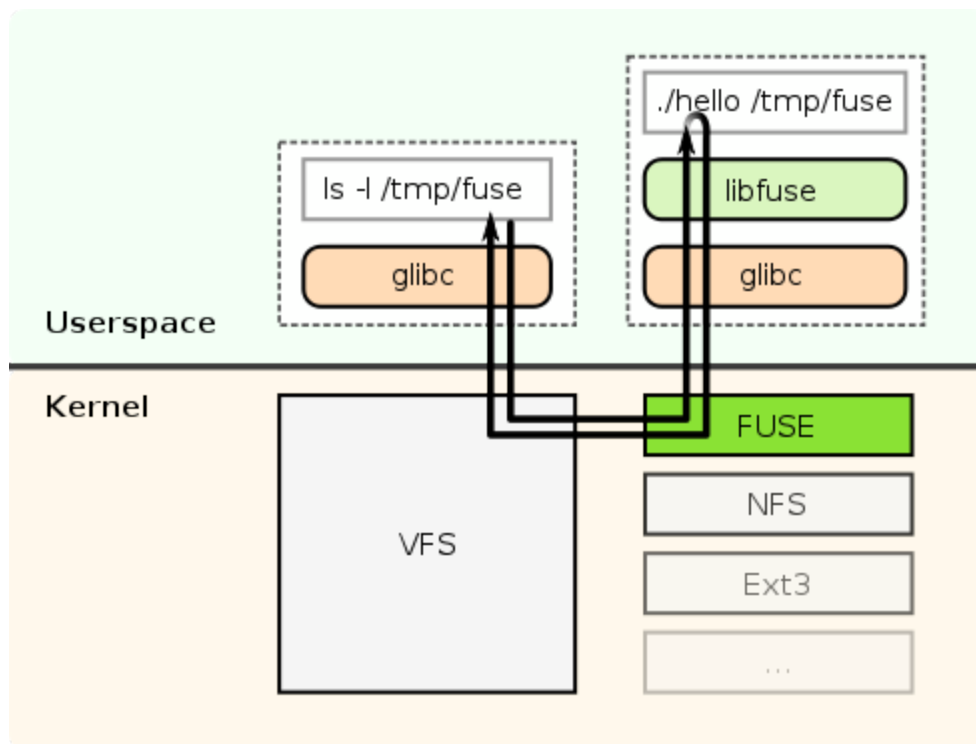


Рисунок 1.8 - Схема работы модуля FUSE

Для операций *монтирования* и *демонтирования* используется специальная утилита **fusermount**, использующая файл конфигурации */etc/fuse.conf*:

fusermount [OPTIONS] MOUNTPOINT

Опция **-u** используется для демонтирования FUSE.

Следует заметить, что многие видные разработчики ОС критикуют FUSE, но эти вопросы выходят за рамки нашей дисциплины.

Замечание

Разделы винчестера, содержащие ФС **ntfs**, монтируются в Linux с типом **fuseblk**.

1.3.3 Дисковые квоты

В заключение темы, рассмотрим вопросы ограничения размеров, которые системно можно накладывать на файловые системы.

Рано или поздно может случиться так, что некоторая ФС на внешнем носителе, подключённая к древовидной структуре ядра ОС, *заполнится и блокирует нормальную работу ОС.*

По умолчанию, ФС *ext2fs* резервирует 5% места для пользователя *root*.

Принято считать, что для нормальной работы ФС требуется *свободное пространство:*

- *не менее 10%* дискового пространства (пространство раздела);
- *не менее наибольшего файла* в файловой системе.

Рекомендуется размещать */home, /opt, /tmp, /var, /usr/local* в разных разделах диска.

Определить свободное пространство диска можно командой *df*:

```
$ df
Файл. система    1К-блоков  Использовано  Доступно  Использовано%  Монтировано в
/dev/sda5         21356772    14841812    5430084      74% /
udev              1669124      12    1669112      1% /dev
tmpfs             670748       780    669968      1% /run
none              5120         0     5120        0% /run/lock
none             1676860      84    1676776      1% /run/shm
none             102400       12     102388      1% /run/user
/dev/sda1         51199120    26771180    24427940     53% /cdrom
$
```

Администратор ОС должен отслеживать наличие достаточного дискового пространства ФС. Для этих целей имеется специальное системное ПО под общим названием **quota**, а ядро ОС должно поддерживать команды этого пакета.

Чтобы включить *квоты* для конкретной файловой системы, нужно в корень ФС поместить бинарные файлы:

- *quota.user* — для персональных квот;
- *quota.group* — для групповых квот.

Управление квотами выполняется с помощью команд:

quotastats	Проверка поддержки квот ядром ОС.
quotacheck	Сканирование заданной ФС и первоначальное создание файлов <i>quota.user</i> и <i>quota.group</i> .
edquota	Редактор квот.
quotaon	Активация настроек квот для ФС.
quotaoff	Деактивация настроек квот для ФС.
quota	Проверка пользователем, установленных для него квот.

2 Лабораторная работа №4

Цель лабораторной работы №4 — практическое закрепление учебного материала по теме «Управление файловыми системами ОС».

Метод достижения указанной цели — закрепление учебного материала, изложенного в первом разделе пособия посредством утилит ОС, а также выполнение заданий, приведённый в данном разделе.

Чтобы успешно выполнить данную работу, студенту следует:

- *запустить с flashUSB* ОС УПК АСУ, подключить личный архив и переключиться в сеанс пользователя *upk*;
- *запустить на чтение* данное пособие и на редактирование личный отчёт;
- *открыть одно или несколько окон терминалов*, причём хотя бы в одном окне терминала открыть *Midnight Commander*, для удобства работы с файловой системой ОС;
- *приступить к выполнению работы*, последовательно пользуясь рекомендациями представленных ниже подразделов.

Замечание

Многие команды ОС студенту ещё не известны, поэтому следует:

- для вывода на консоль руководства по интересующей команде, использовать: **man имя_команды**;
- для выяснения существования команды, ее доступности и местоположения, использовать: **command -v имя_команды**;
- для уточнения правил запуска конкретной команды, можно попробовать один из вариантов: **команда --help** или **команда -h** или **команда -?**.

В процессе выполнения лабораторной работы студент заполняет личный отчёт по каждому изученному вопросу!

2.1 Типы, имена и узлы устройств

Прочитайте и усвойте материал подраздела 1.1: «Устройства компьютера».

Исследуйте содержание директории */dev*.

Исследуйте содержание директории */proc*.

С помощью утилиты *man* изучите утилиты *mknod*, *ls* и *grep*.

Научитесь в терминале выводить характеристики узла, зная его имя.

2.2 Структура винчестера и файловые системы

Прочитайте и усвойте учебный материал пунктов 1.1.1-1.1.5.

Закрепите изученный материал посредством практического использования утилит: *fdisk*, *mkfs*, *mknod*, *mount* и *umount*.

2.3 Стандартизация структуры ФС

Прочитайте и усвойте материал подраздела 1.2: «Стандартизация структуры ФС».

Запустите в окне терминала файловый менеджер **Midnight Commander** и с помощью него изучите структуры ФС всех трёх уровней и директории `/var`.

В окне терминала, усвойте работу команд (утилит) `pwd`, `cd`, `ls`, `cat`.

В домашней директории пользователя `upk`, усвойте работу утилит `mkdir` и `rmdir`.

2.4 Модули и драйверы ОС

Прочитайте и усвойте учебный материал пунктов 1.2.1-1.2.2.

В окне терминала исследуйте ветвь дерева директории `/lib/modules`.

В окне терминала, усвойте работу утилит `insmod`, `rmmmod`, `lsmod`, `modinfo` и `dmesg`.

Подробно изучите группировку и назначение функций системных вызовов ОС, представленных таблицей на рисунке 1.9.

2.5 Концепции работы с устройствами

Прочитайте и усвойте учебный материал подразделов 1.3 и пункт 1.3.1.

Разберитесь в основных концепциях работы с устройствами (узлами) ОС.

С помощью руководства `man` и файла `/etc/fstab` изучите способ задания конфигурации блочных устройств ОС.

Сравните структуру файла `/etc/fstab` с выводом утилиты `mount`.

На примерах подраздела 1.12, освоите правильное использование утилит `mount` и `umount`. Особое внимание уделите атрибутам (опциям) монтирования.

2.6 FUSE и другие специальные ФС

Прочитайте и усвойте учебный материал пунктов 1.3.1-1.3.3.

На учебных примерах разберитесь с особенностями монтирования изученных ФС.

На практике освоите работу с утилитами: `df`, `du` и `lsdf`.

Замечание

Примеры, приведённые в подразделе 1.3, являются чисто демонстрационными и не могут быть напрямую реализованы в ОС УПК АСУ.

2.7 Подключение рабочей области пользователя `upk`

На примере файла сценария `/etc/upkasu/mount-upk.sh`, разберитесь и опишите в отчёте алгоритм и функции монтирования рабочей области пользователя `upk`.

Список использованных источников

- 1 Резник В.Г. Операционные системы. Самостоятельная и индивидуальная работа студента по направлению подготовки бакалавра 09.03.03. Учебно-методическое пособие. – Томск, ТУСУР, 2016. – 13 с.
- 2 Гордеев А.В. Операционные системы: учебное пособие для вузов. – СПб.: Питер, 2004. – 415 с.
- 3 Таненбаум Э., Бос Х. Современные операционные системы. 4-е изд. — СПб.: Питер, 2015. — 1120 с.: ил. — (Серия «Классика computer science»). ISBN 978-5-496-01395-6
- 4 Резник В.Г. Учебный программный комплекс кафедры АСУ на базе ОС ArchLinux. Учебно-методическое пособие. – Томск, ТУСУР, 2017. – 38 с.
- 5 Резник В.Г. Операционные системы. Тема 1. Назначение и функции ОС. Учебно-методическое пособие. – Томск, ТУСУР, 2017. – 33 с.
- 6 Резник В.Г. Операционные системы. Тема 2. BIOS, UEFI и загрузка ОС. Учебно-методическое пособие. – Томск, ТУСУР, 2017. – 30 с.
- 7 Резник В.Г. Операционные системы. Тема 3. Языки управления ОС. Учебно-методическое пособие. – Томск, ТУСУР, 2017. – 38 с.