ОГЛАВЛЕНИЕ

ОГЛАВЛЕНИЕ 1
Введение. 2
I. Файловая система ОС Unix 2
1.1. Основные понятия. 2
1.2. ФС ОС UNIX - вводные замечания. 2
1.3. Системная организация файловой системы. 4
1.3.1. Структура файловой системы на диске. 4
1.3.2 Индексный дескриптор файла (ИД). 6
1.4. Основные типы файлов и их свойства. 8
1.4.1. Предварительное замечание. 8
1.4.2. Рабочие файлы. 8
1.4.3. Каталоги. 8
1.4.4. Специальные файлы (файлы устройств). 10
1.4.5. Другие типы файлов. 12
1.4.6 Многопользовательский режим. Защита файлов, права доступа. 12
1.4.7.Особенности работы с системными файлами. 15
1.6. Организация обмена данными с файлами 16
1.6.1. Низкоуровневый ввод/вывод. 16
1.6.2 Открытие/закрытие файла. 17
1.6.3 Чтение/запись данных. 18
1.6.4. Данные ассоциированные с операционной системой. 18
1.6.5. Данные ассоциированные с процессом. 19
1.6.6. Взаимодействие с устройствами. 20
1.6.7. Стандартная библиотека ввода/вывода. 21
1.7.Монтируемость файловой системы. 21
2. Файловая система Windows NT. 22
2.1. Особенности ОС Windows NT, влияющие на организацию ввода-вывода 22
2.1.1. Объектная модель и контроль доступа в Windows NT. 28
2.1.2. Организация ввода-вывода. 34
2.2. Файловая система NTFS. 37
2.2.1. Основные особенности NTFS и модель функционирования 37
2.2.2. Общие принципы организации на диске 40
2.2.3. Восстанавливаемость 43

Введение.

I. Файловая система ОС Unix

1.1. Основные понятия.

Файловая система (ФС) - часть операционной системы, представляющая собой совокупность организованных наборов данных, хранящихся на внешних запоминающих устройствах и программных средств, гарантирующих именованный доступ к этим данным и их защиту. Данные называются ***файлами***, их имена - ***именами файлов***.

Файловые системы можно классифицировать по степени персонификации доступа к содержимому файлов, соответственно могут быть:

- однопользовательские файловые системы;
- многопользовательские файловые системы.

Признаком однопользовательской **ФС** может служить система, в которой не регламентируется доступ к содержимому файлов от имени любого пользователя. Примером организации таких файловых систем может служить **MS DOS**. В данной системе отсутствуют возможности какой-либо персонификации работы пользователей в том числе и с файловой системой. Другим примером таких систем может служить операционная система **WINDOWS-95,** в которой имеются зачаточные средства идентификации и регистрации пользователей. Однако, эти средства не распространяются на файловую систему, она остается однопользовательской, т.е. информация из любого файлов доступна всем.

Многопользовательские файловые системы это атрибут развитых операционных систем, предусматривающих работу только идентифицированных системой пользователей. Для многопользовательских файловых систем основным свойством является наличие защиты данных, содержащихся в файлах от несанкционированного доступа. Примерами операционных систем, обладающих подобными файловыми системами являются **UNIX, WINDOWS-NT** и ряд других.

1.2. ФС ОС UNIX - вводные замечания.

Файловая система операционной системы **UNIX** является примером многопользовательской иерархической файловой системой с трехуровневой организацией прав доступа к содержимому файлов .

Файловую систему можно представить в виде дерева (Рис.1), корнем которого является “начальный каталог”. Будем обозначать его символом наклонная черта **‘/’.** Узлами дерева, являются каталоги (следует отметить что, с точки зрения организации **ОС UNIX**, каталоги так же являются файлами системы). Листом дерева может быть либо файл (в традиционном понимании), либо пустой каталог.

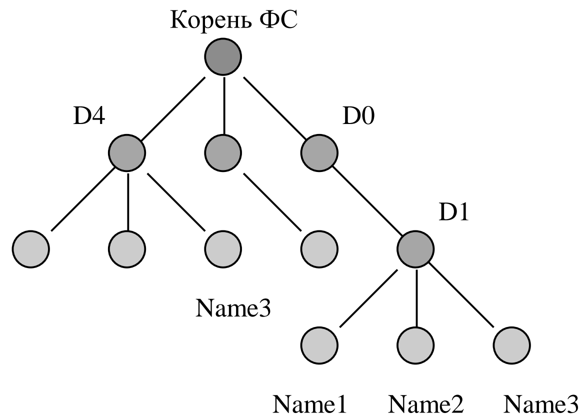


Рис.1 Иерархическая организация файловой системы

Ввиду иерархической организации файловой системы, представленной в виде дерева, принято многоуровневое именование файлов. В любом отдельно взятом каталоге имена файлов уникальны. Имя файла может рассматриваться относительно каталога, в котором он находится. Полное имя - это указание всего пути от корня к данному файлу. Если имя начинается с **/**, то считается, что далее идет полное имя. Далее указываются через / все каталоги, находящиеся на пути от корня до этого файла. Пример: **/D0/D1/Name3** или **/D4/Name3** - разные файлы. Главное ограничение заключается в том, чтобы не совпадали имена файлов в одном каталоге.

В операционной системе **Unix** могут быть следующие категории пользователей (Рис.2):

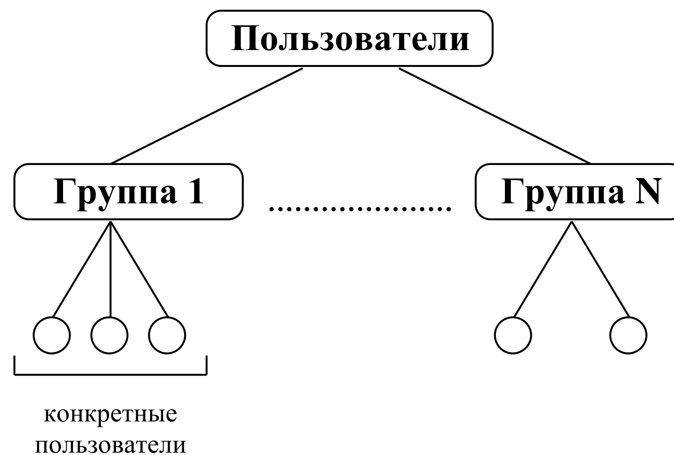


Рис.2 Иерархия пользователей системы

Из рис.2 видно, что определены следующие уровни: всех пользователей системы; групп пользователей; конкретных пользователей. Для каждого файла определено понятие “владелец файла”. Права доступа к каждому файлу организованы по схеме, соответствующей общей организации пользователей: права доступа к содержимому файлов для владельца файла; права доступа к содержимому файлов для группы, к которой принадлежит владелец файла; права доступа для всех остальных пользователей операционной системы.

1.3. Системная организация файловой системы.

1.3.1. Структура файловой системы на диске.

Для любой операционной системы определено понятие ***системное внешнее запоминающее устройство.*** Это устройство к которому обращается ***аппаратный загрузчик*** для загрузки операционной системы. Для операционной системы **Unix** файловая система также располагается на ее системном устройстве. Рассмотрим организацию данных, размещенных на системном **ВЗУ**, которые образуют файловую систему.

В системе принято разбиение пространства внешних запоминающих устройств, на которых может размещаться файловая система или ее части на блоки - части равного для данной системы размера. Размер блока может варьироваться от версии к версии системы **Unix** или быть параметром настройки системы (блок может содержать 256, 512, 1024 байта). Для простоты изложения будем считать размер блока равным 512 байт. Итак, рассмотрим структуру файловой системы на системном устройстве (Рис.3).

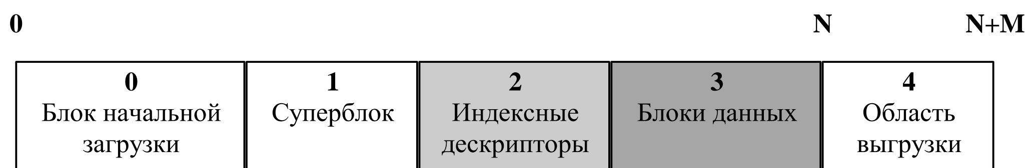


Рис.3 Распределение файловой системы на системном **ВЗУ**.

Будем считать, что системное **ВЗУ** размещается с нулевого блока реального устройства (на самом деле возможно смещение).

В нулевом блоке (в блоке с номером ноль) системного устройства всегда находится ***программный загрузчик*** - программа, к которой обращается аппаратный загрузчик вычислительной машины и которая обеспечивает загрузку операционной системы, размещенной на данном системном устройстве. Это верно, практически для всех вычислительных систем. В нотации операционной системы **Unix** нулевой блок называется ***блоком начальной загрузки.*** Последовательность действий при старте вычислительной системы следующая.

1. Запускается аппаратный загрузчик - программа, размещенная в постоянном запоминающем устройстве **(ПЗУ)** вычислительной машины. Аппаратный загрузчик обеспечивает считывание содержимого блока начальной загрузки в память машины и передает управление на фиксированный адрес считанных данных - ***адрес входа в программный загрузчик.*** Обычно, этот адрес фиксирован для вычислительных машин данного типа и служит адресом входа для программных загрузчиков всех операционных систем эксплуатируемых совместно с данной ЭВМ.

2. Программный загрузчик является компонентой конкретной операционной системы, он “знает” структуру системного устройства и обеспечивает полный запуск операционной системы.

Следующий блок системного устройства операционной системы **Unix** - это ***суперблок*** файловой системы. В суперблоке размещается глобальная информация о параметрах настройки файловой системы и о ее текущем состоянии, а именно:

- количество ***Индексных Дескрипторов (ИД),*** (основных элементов, описывающих атрибуты файла). Это количество формируется во время создания файловой системы. Понятие **ИД** будет подробно рассмотрено ниже.
- максимальный номер блока **N**, который может быть использован для хранения данных. Также формируется во время создания **ФС.**
- число свободных блоков **S_NFREE,** номера которых находятся в специальном массиве **S_FREE[50].**
- число свободных **ИД - S_NINODE**, номера которых содержатся в специальном массиве **S_INODE[100].**
- некоторая другая служебная информация (например, время последней модификации суперблока и т.д.) **.**

Так как суперблок хранит важную информацию в системе хранится несколько его копий.

Следующая область системного устройства предназначен для хранения индексных дескрипторов - ***область индексных дескрипторов***, длина которого определяется их количеством.

Далее следует ***пространство для хранения данных*** на диске.

Все свободное пространство диска представляет из себя связанный список (рис.4). Приведем алгоритм работы со свободным адресным пространством (алгоритм поиска свободных блоков на диске).

> Если **S_NFREE!=0,** то выбирается блок из массива **S_FREE[S_NFREE].** Далее **S_NFREE:=S_NFREE-1.** В противном случае, т.е., когда **S_NFREE=0**, первый элемент массива **S_FREE** указывает на блок, содержащий список еще 49 свободных блоков, который считывается в суперблок. Список всех свободных блоков заканчивается в том случае, когда в первом элементе массива **S_FREE** появляется ноль (список исчерпан).
>
> При освобождении блоков соответствующие номера записываются в массив **S_FREE[S_NFREE+1]** и **S_NFREE:=S_NFREE+1.**
>
> После достижения **S_NFREE** значения 50, содержимое массива **S_FREE** копируется в свободный блок, а его адрес записывается в элемент **S_FREE[0].**

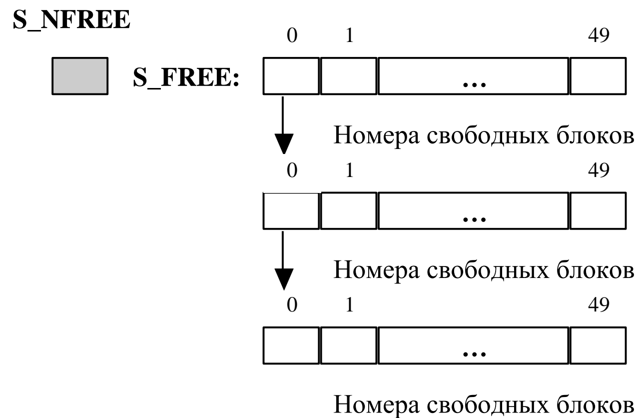


Рис.4 Список свободных блоков.

В силу того, что суперблок практически всегда находится в памяти, имеется возможность быстро без обращения к диску занять до 49 свободных блоков. С другой стороны, освобождаемые блоки легко подсоединяются к списку свободных. Рассмотренный алгоритм работы со свободным пространством **ФС** предполагает, что первым использованным блоком из числа свободных будет последний освободившийся.

Последняя область - ***область выгрузки (свопинга).*** Используется **ОС** для размещения образов памяти откаченных процессов, а также для размещения процессов, файлы которых специфицированы ***t-битом***. В современных реализациях для области свопинга может быть организована специализированная **ФС.**

1.3.2 Индексный дескриптор файла (ИД).

ИД - это основной элемент, описывающий атрибуты файла. Каждый файл определяется **ИД**, который содержит всю необходимую информацию о данном файле. **ИД** состоит из некоторого количества полей, в частности:

- поле, содержащее биты, характеризующие тип файла, его привилегии и код защиты. Например, восьмеричная маска 040000 - означает, что файл является каталогом, 060000 - означает, что это блок-ориентированный специальный файл, 000020 - дано разрешение на запись для владельца файла и так далее.
- поле, содержащее количество ссылок из каталогов к **ИД.**
  Отметим, что ***каталоги содержат список имен файлов и ссылки*** на **ИД**. Одному **ИД** могут соответствовать несколько имен файлов (эти имена могут встречаться в произвольных местах **ФС**). Каждая новая ссылка к **ИД,** отмечается в данном поле, что позволяет **ФС** следить за занятостью файла. В корректной **ФС** должно быть полное соответствие между числом ссылок в данном поле и числом записей, относящихся к данному **ИД**. Как только этот счетчик становится равным нулю, **ИД** освобождается, а дисковое пространство может быть использовано для записи других файлов.
- два поля, предназначенных для определения привилегий доступа к файлу. Здесь содержатся коды идентификации пользователя и его группы, создавшей файл.
- поле, служащее для записи длины файла.
- поле, фиксирующее время последнего обращения к файлу.

- два поля, предназначенные для фиксации времени создания и последней модификации файла.
- поле, предназначенное для адресации данных файла на диске.

Расположение файла задается списком его блоков. Это снимает проблемы непрерывных файловых систем, т.е. систем, где блоки файла располагаются последовательно. Таким образом реально блоки файла могут быть разбросаны по диску, но логически они образуют цепочку, содержащую весь набор данных. Ключом, задающим подобное расположение служит список из 13 номеров блоков на диске, хранящихся в **ИД**. Первые десять указывают на десять блоков некоторого файла. Если файл занимает более 10 блоков, то 11 элемент указывает на косвенный блок, содержащий до 128 адресов дополнительных блоков файла (это еще 70656 байт). Большие файлы используют 12 элемент, который указывает на блок, содержащий 128 указателей на блоки, каждый из которых содержит по 128 адресов блоков файла. Еще в больших файлах аналогично используется 13 элемент. Трехкратная косвенная адресация позволяет создавать файлы длиной (10+128+128*128+128*128*128)*512 байт. Таким образом, если файл меньше 512 байт, то необходимо одно обращение к диску, если длина файла находится в пределах 512-70565 байт, то - два и так далее. Приведенный способ адресации позволяет иметь прямой и быстрый доступ к файлам. Эта возможность также усиливается кэшированием диска, позволяющим хранить в памяти наиболее используемые блоки . Важно отметить, что при ***открытии файла*** соответствующий **ИД** считывается в память и системе становятся доступны все номера блоков данного файла. Кроме того, для одного и того же файла, открываемого несколько раз, в памяти находится только один **ИД.** Система фиксирует число открытий данного файла и, когда этот счетчик обнуляется, резидентный образ **ИД** переписывается на диск. Если при этом изменений в файле не было и не модифицировался **ИД,** то запись не выполняется. Указанные особенности существенно влияют на эффективность файловой системы.

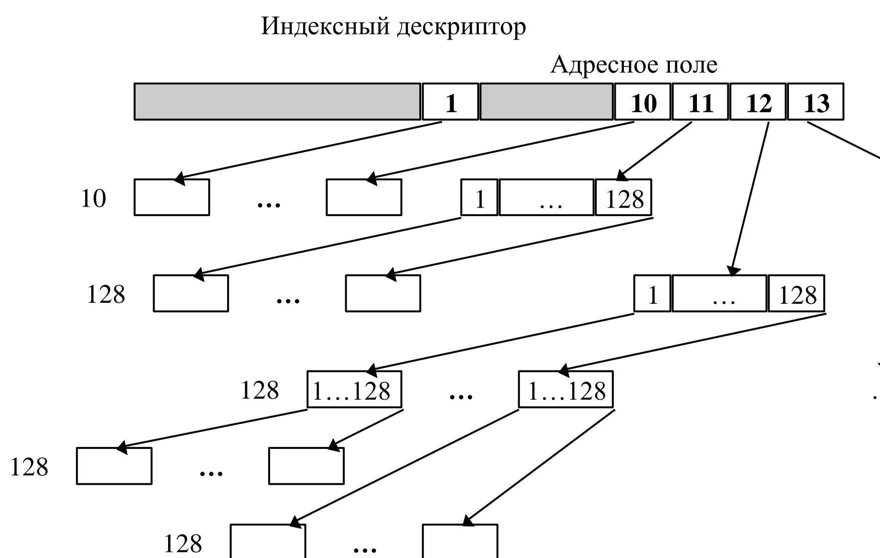


Рис.5 Адресация данных файла.

Отметим, что алгоритм для поиска свободных **ИД** в массиве **S_INODE**, суперблока, похож на алгоритм для поиска свободных блоков на диске. Отличие состоит в том, что после исчерпания массива, он дополняется путем линейного поиска свободных **ИД** в блоке, где хранятся свободные **ИД.** Т.е. не происходит считывание заранее подготовленных записей. Это оправдано, ибо создание и удаление файла происходит гораздо реже, нежели занятие и освобождение новых блоков файла.

1.4. Основные типы файлов и их свойства.

1.4.1. Предварительное замечание.

Система не накладывает на информацию, хранимую в файле, никаких ограничений или структурных требований.

1.4.2. Рабочие файлы.

Данные файлы размещаются на диске и содержат ту информацию, которую занес в него пользователь или которая получилась в результате работы команд системы, трансляторов, редакторов и так далее. Все файлы так или иначе - двоичные. При этом в зависимости от интерпретации файл может содержать исходный текст программы, главу книги , готовый к исполнению код, словом все, что необходимо сохранить для дальнейшего использования. Важно то, что тип файла определяется не файловой системой.

1.4.3. Каталоги.

Каталог - это файл, осуществляющий связь между именами файлов и собственно файлами, создавая тем самым структуру **ФС**. В каталог могут входить имена рабочих файлов, каталоги и специальные файлы.

Выше говорилось, что существует соглашение об именовании файла, т.е. полное имя представляет собой путь от корневого каталога через имена каталогов до требуемого файла. При создании каталога в нем образуются два стандартных файла. Имя файла **'.'**в каждом каталоге связано с самим каталогом, а имя **'..'** с родительским каталогом. С помощью имени **'..',** имеющемся в каждом каталоге, можно сделать шаг вверх в иерархии **ФС**. Каталог считается пустым если он содержит только эти два файла.

Внутреннее устройство каталога представляет собой массив структур, состоящих из двух элементов. Первый элемент - номер индексного дескриптора, а второе - имя файла. Таким образом, каталог - это массив, состоящий из списка имен файлов и ссылок на **ИД.**

Видно, что при такой реализации имя файла "отделено" от других его атрибутов. Это позволяет, в частности, один и тот же файл внести в несколько каталогов. При этом данный файл может иметь разные имена в разных каталогах, но ссылаться они будут на один и тот же **ИД**, который является ключом для доступа к данным файла. При обсуждении понятия **ИД** говорилось, что каждая новая ссылка к **ИД** отмечается в специальном поле. Рассмотрим пример, иллюстрирующий связь файлов с каталогами (рис.6-7).

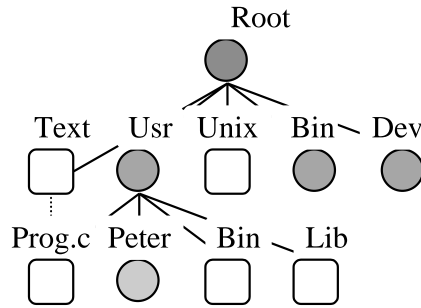


Рис.6 Фрагмент файловой системы

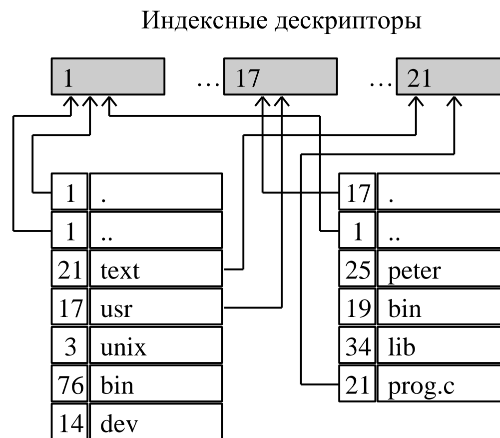


Рис. 7. Связь файлов с каталогами.

В приведенном примере необходимо подчеркнуть следующие моменты. Первый - **ИД** с номером 1 всегда содержит атрибуты корневого каталога. Файлы с именами **“.”** и **“..”** в каталоге **root** ссылаются на ИД с номером 1, т.е. сами на себя. Имена **text** и **prog.c** ссылаются на один и тот же **ИД**, т.е. на один и тот же файл. В частности, если задать команду удаления файла
rm prog.c,
то реально произойдет удаление не файла, а строки в соответствующем каталоге (т.е. связи). Если задать далее команду
rm text,
то произойдет ликвидация последней связи, а, следовательно, **ИД** и самого файла. Далее, как отмечалось выше, **ИД** объявляются свободными, а блоки, занятые данными файла, объявляются свободными.

Траектория, которая задается полным именем, представляет собой переходы между каталогами и **ИД**. Рассмотрим, к примеру, полное имя **‘../a/b’**. Это имя есть путь из текущего каталога в его родительский каталог, далее в подкаталог **‘a’** родительского каталога и , наконец, к файлу с именем **‘b’** из каталога **‘a’**. Для прохождения этого пути, система проделает следующие действия.

1. Выбирает из контекста процесса **ИД** текущего каталога.
2. Используя информацию из этого **ИД**, ищет в текущем каталоге имя **‘..’** и получает номер его **ИД**.
3. Выбирает **ИД** **‘..’**

4. Используя информацию из этого **ИД**, ищет в родительском каталоге файл **‘a’** и получает номер его **ИД.**
5. Выбирает ИД **‘a’.**
6. Используя информацию из этого **ИД,** проводит поиск в каталоге **‘a’** файла с именем **‘b’** и получает номер его **ИД.**
7. Выбирает **ИД .**
8. Обращается к файлу **‘b’**.

Приведенный выше пример демонстрирует, что самый обычный доступ к файлу требует определенных затрат - это расплата за преимущества применения иерархической **ФС** (рис.7). Однако, реально прослеживание цепочек по полному имени проводится относительно редко, чаще осуществляются обращения к ранее обнаруженным файлам .

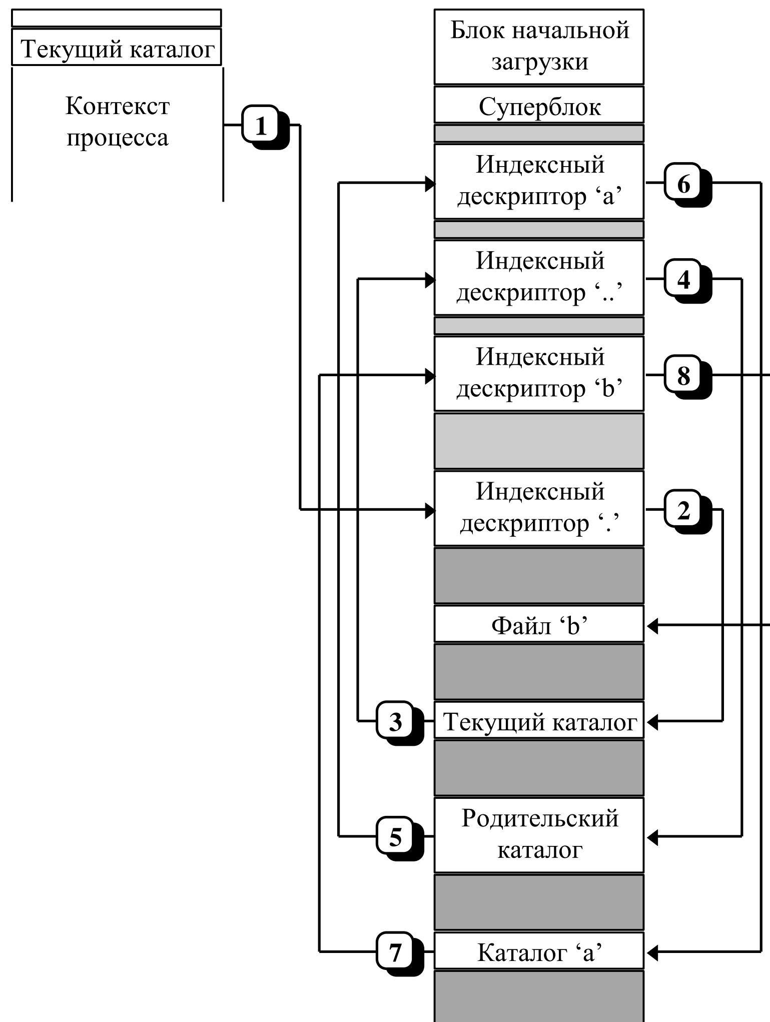


Рис.8 Прослеживание цепочки полного имени.

1.4.4. Специальные файлы (файлы устройств).

В **ОС Unix** достигнута унификация доступа к внешним устройствам за счет использования **ИД**, которые обеспечивают связь между иерархической структурой **ФС** с драйверами. Каждое внешнее устройство ввода/вывода

связано с одним именем, которое хранится в каталоге **/dev**. Такая интерпретация подобных устройств позволяет синтаксически идентично работать как с внешними устройствами, так и с “обычными” файлами, применяя при этом тот же механизм защиты. Система обнаруживает отличие обычного файла от специального только после анализа **ИД**, к которому ссылается запись в каталоге. **ИД,** в свою, очередь содержит информацию о классе устройства (блок-ориентированное или байт-ориентированное) и его номере. Блок-ориентированные устройства - это диск ,магнитная лента, а байт-ориентированные устройства - терминал, принтер и так далее. К примеру, при обсуждении понятия **ИД** говорилось, что в первом поле определяется тип файла. Если там находится код 020000, то это байт-ориентированное устройство, а если 060000 - блок-ориентированное.

Таким образом специальные файлы обеспечивают интерфейс с внешними устройствами. Они условны в том смысле, что они не хранят какую-либо информацию, а используются в качестве канала для доступа к внешним устройствам (в ядре обращение к такому файлу преобразуется в машинные команды обращения, к примеру, к **МЛ**, как если бы программа читала файл **/dev/mt0**, выдавая содержимое **МЛ**, подключенной к устройству).

Итак вся информация об устройстве и его драйверах находится в индексном дескрипторе. А именно в его поле **di_mode** (1 поле **ИД**) указывается тип устройства (байт-ориентированное или блок-ориентированное). В поле **i_addr[0]** находится структура данных с двумя полями:

- **d_ major,** определяющее тип устройства и выбор драйвера ввода/вывода и
- **d_minor** , передающееся драйверу в качестве параметра.

Доступ системы к драйверу соответствующего устройства осуществляется через две системные таблицы **cdevsw** и **bdevsw**, содержащие указатели на подпрограммы драйверов (первая для байт-ориентированных, вторая для блок-ориентированных устройств). Выбор таблицы производится по значению битов в поле **di_mode**, а внутри таблиц драйвер выбирается на основе типа устройства **d_major**. Любой файл, к которому производится обращение, ***должен быть открыт*** или создан системными вызовами ***open*** или ***creat*** (более подробно см.ниже).

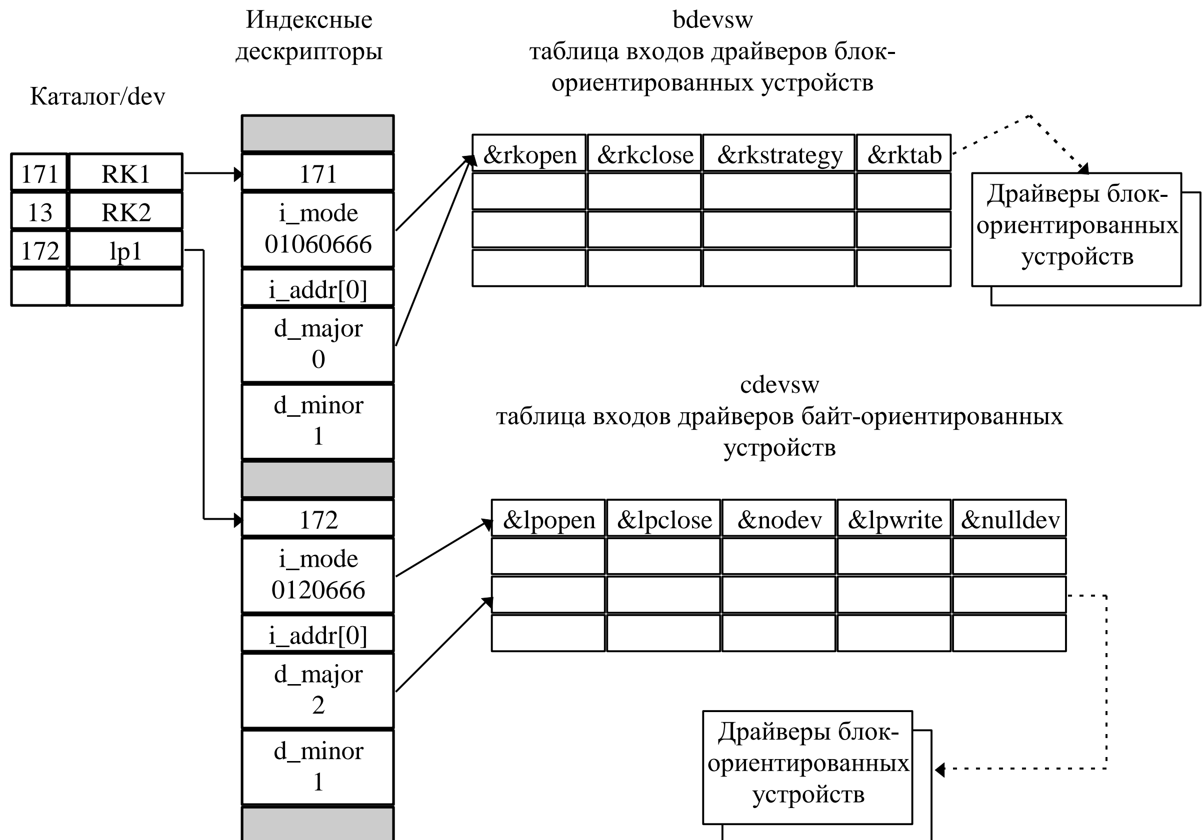


Рис.9 Структура системы ввода/вывода

1.4.5. Другие типы файлов.

В различных версиях **ОС Unix** помимо перечисленных типов файлов могут использоваться и другие типы, а именно: доменные гнезда(sockets), именованные каналы, жесткие ссылки и символические ссылки. Подробное рассмотрение этих типов не входит в цели данного пособия.

1.4.6 Многопользовательский режим. Защита файлов, права доступа.

При создании файлу присваивается код защиты (например, используя системный вызов **creat**) - слово, биты которого характеризуют тип файла и права доступа процессов к нему. Код защиты находится в **ИД** и занимает младшие девять битов этого слова.

С другой стороны, существуют 3 категории пользователей:

1. Владелец
2. Группа, к которой принадлежит владелец
3. Остальные

Для каждой из этих групп можно определить одно из следующих прав: право на чтение (**R,** восьмеричный код 400)**,** право на запись (**W,** код 200) и право на выполнение (**X ,**код 600) или их комбинацию.

Например:

Владелец **RWX**

Группа **R - X**

Остальные **R - -**
т.е. код защиты будет следующим: **RWXR-XR - -.**

Некоторые файлы помечаются битом сохранения **t-bit** (**sticly-bit,** восьмеричный код 1000)**.** В этом случае после того как этот файл был вызван в качестве процесса и проработал, он остается в пространстве свопинга . Это делается для того, чтобы впоследствии не затрачивать время на сборку файла с диска по частям. Файлу режим сохранения придается администратором системы. Режим сохранения может задаваться лишь для небольшого числа процессов, так как пространство свопинга ограничено.

Иногда приходится применять другие способы защиты для обеспечения требуемого режима доступа к файлам. Речь идет об использовании, так называемого **s-bit**а. Владелец файлов может установить такой режим, в котором другие пользователи имеют возможность назначать собственные идентификаторы режима. Это указывается символом **"s"** на месте признака, идентифицирующего разрешение владельцу запускать файл на выполнение **"x".** Аналогичным образом разрешение установки идентификатора группы позволяет лицу, выполняющему программу, приобретать привилегии члена группы, в которую входит владелец программы. Фактически установка группового идентификатора - это средство временного перевода пользователей в категорию системных. Соответствующие файлы идентифицируются символом **"s"** на месте признака **"x",** указывающего на разрешение членам группы запускать файл на выполнение.

Выше отмечалось, что при работе с каталогом его владельцу, члену соответствующей группы, и всем остальным пользователям разрешается чтение, запись и выполнение. Однако, эти разрешения интерпретируются не так, как для обычных файлов. Разрешение на чтение из каталога означает, что разрешено открытие каталога и чтение из него. Разрешение на запись предоставляет возможность создавать и уничтожать файлы. Разрешение на выполнение свидетельствует о том, что система может выполнять поиск в каталоге с целью обнаружения какого-либо файла. Если вместо простого имени используется составное, то в каждом из указанных в нем каталогов выполняется поиск имени файла, стоящего следующим в составном имени.

Примеры.

Пример 1.

Команда **file** [имена] , позволяющая обоснованно догадаться о типе файла (ибо тип файла не определяется **ФС**).

file /wert ter1.test tr4.c

/wert - directory
tet1.test - ascii file
tr4.c - c program text

Команда **file** читает первые несколько сотен байтов файла и пытается понять его тип. Если это исполняемый файл, то файл помечается некоторым, зависящим от системы, специальным числом (**magic code**). Если это текстовый файл, то указание может быть упрятано более глубоко в файле. Поэтому

команда **file** будет отыскивать строки, подобные #include, чтобы распознать текст программы на языке **C** и так далее.

Пример 2.

Рассмотрим команду **ls [-флаги] [имя]** - вывод списка имен текущего каталога. В частности **ls -l** выводит список файлов, содержащий следующую информацию.

ls -l /usr/you

Результатом будет таблица, состоящая из строк вида:

-rwxrwxrwx [число связей] [имя владельца] [имя группы] [размер в байтах] дата модификации [имя файла]

Например:

ls -l /udd/users/ter

```
- rw- rw -rw-   1  ter   people    35   Jan  5 96   tr
drwxrwxrwx      2  ter   people   226   Apr  3 97   TEST/
```

Первый символ обозначает: если " **-** " , то это обычный файл, если **" d "** - каталог. Далее три триады говорят о правах доступа к файлу (об этом говорилось выше). Первая триада - права доступа для пользователя **(ter),** вторая триада - права доступа для группы **(people),** последняя триада для всех остальных. Цифры 1 и 2 соответственно обозначают количество ссылок к **ИД** по всей **ФС**. Числа 35 и 226 - размер файлов в байтах. Далее дата последней модификации. Наконец, последняя информация говорит о том, что **tr** - файл, находящийся в директории **ter,** а **TEST** - каталог нижнего уровня.

Пример 3.

ls -l /dev

```
crw - -w - - w-     1   root   0 ,   0   Jun   5   23:07   console
brw - rw - rw -     1   root   1,   64   Jul   8   23:08   mt00
crw - - w - - w -   1   root   1,    0   Oct   3   21:23   tty0
crw - - w - -w -    1   root   1,    1   Oct   4   21:43   tty1
```

Картина несколько отличается от предыдущей. В данном случае **ИД** содержит внутреннее имя устройства - первый символ обозначает тип устройства (байт-ориентированное - **"c"** или блок-ориентированное - **" b"**) и пару чисел, которые называются верхним и нижним числом устройства (**d_minor, d_magor**). Верхнее число обозначает его тип, а нижнее характеризует различные экземпляры устройств данного типа. В примере две нижние записи описывают два порта одного контроллера терминала, поэтому они имеют одно и то же верхнее число и разные нижние числа. Короче говоря, в данном случае (в случае специального файла) **ИД** хранит не список блоков памяти диска, а внутреннее имя устройства и его тип.

Пример 4.

Рассмотрим команду

chmod права доступа [имена файлов],

позволяющую изменять права доступа к файлам. Ниже приведен один из возможных вариантов использования этой команды.

chmod 226 /udd/users/ter/tr,
где 226 - восьмеричная маска, задающая код защиты.

1.4.7.Особенности работы с системными файлами.

Система **Unix** предоставляет удобную форму работы с системными файлами. Речь идет о таких файлах как……..При входе в систему пользователь вводит имя. Получив это имя команда **login** ищет в учетном файле **/etc/passwd** строку, содержащую это имя. Имя - это входной идентификатор пользователя - **login-id.** Если в этой строке присутствует пароль, **login** запрашивает этот пароль с терминала. Система распознает пользователя по идентификатору пользователя - **uid.** Помимо **uid** пользователю приписывается идентификатор группы - **group-id**. Файл **/etc/passwd** - файл паролей, представляет из себя обычный текстовый файл. Данный файл содержит всю информацию, связанную со входом каждого пользователя в систему. Поля в файле паролей разделяются двоеточием и расположены следующим образом:

login-id*:* ***зашифрованный пароль*****: uid: group-id:** любая информация**:** начальный_каталог**: shell**

5.Структура файловой системы с точки зрения пользователя

Существует набор файлов и каталогов с известными (предопределенными) именами. Ñистема знает, что в этих файлах находится соответствующая информация(рис.10).

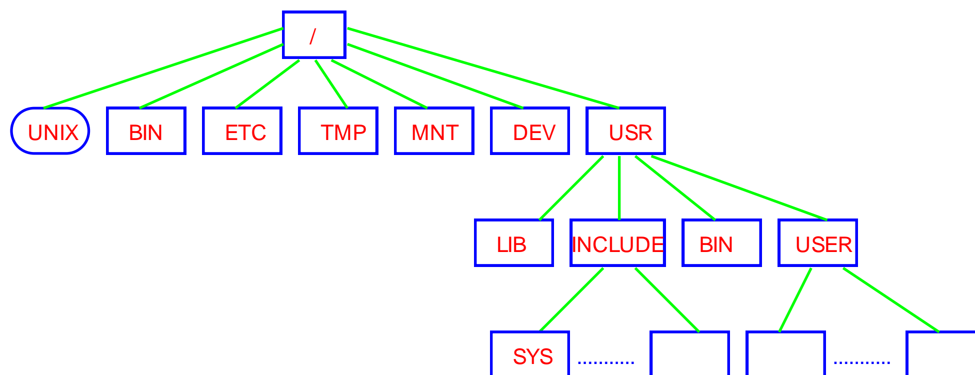


Рис.10. Содержание корневого каталога.

/unix - файл загрузки ядра **ОС**. Выше говорилось, что при включении машины аппаратный загрузчик запускает программный загрузчик в **“ 0 “**блоке, а тот в свою очередь запускает ядро.
/bin - файлы реализующие общедоступные команды системы. Необходимо отметить, что в **ОС Unix** нет набора команд системы. Пользователь это может сделать сам. Это достигается за счет двух моментов.

- в **ОС Unix** запуск исполняемой программы осуществляется по имени файла (т.е. набором имени файла).
- существует стандарт на прием и обработку параметров, заданных в командной строке вида :

<имя файла> параметр1 параметр2 …,

что обеспечивает доступ к строкам из **С**-программ.
/etc - файлы, используемые для управления системой (passwd, mount, unmount).
/tmp - хранятся временные данные системы (до первой перезагрузки);
/mnt - директория, к которой монтируется монтируемая **ФС** (см.ниже);
/dev - каталог, содержащий файлы устройств;
/usr - каталог, связанный с работой пользователя в системе:

/usr/lib - системы программирования, редакторы и т.д. Например, здесь находятся препроцессор **С (/lib/cpp),** библиотека стандартных функций **С (/lib/libc.a);**
/usr/bin - второй уровень команд системы (программы пользователя);
/**usr/user** - домашние каталоги различных пользователей;
/usr/include - файлы макроопределений **С**-программ;
/usr/include/sys - системные файлы макроопределений **С**-программ;

В частности, результатом команды **ls /** будет следующее

unix
bin
etc
tmp
mnt
dev
usr

1.6. Организация обмена данными с файлами

1.6.1. Низкоуровневый ввод/вывод.

Мы рассмотрим основные принципы организации, так называемого, ***низкоуровнего*** обмена с файлами. В операционной системе **UNIX** для этих целей используются ***системные вызовы***, осуществляющие непосредственное обращение к средствам **ОС** и, в частности, к ядру (Рис.11). Рассмотрим это взаимодействие с точки зрения пользовательского интерфейса, а точнее набора функций предоставляющего возможности организации этого взаимодействия.

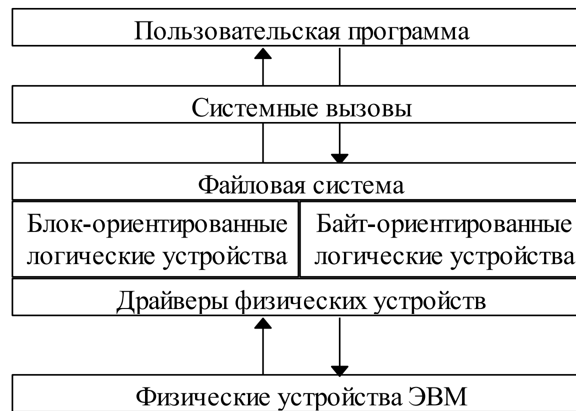


Рис.11 Структура системы управления вводом/выводом

1.6.2 Открытие/закрытие файла.

Операционная система предоставляет процессу доступ к содержимому файла после того как он был ***открыт.*** Открытие файла - это своего рода регистрация процесса в операционной системе на доступ к содержимому фала.

Для выполнения операции открытия существующего файла используется функция

int open(char *name, int flag).

Параметрами данной функции служат:

- **name** - указатель на строку, содержащую полное имя открываемого файла;
- **flag -** значение, специфицирующее режим открытия файла (открытие файла только на чтение или только на запись).

Функция возвращает целое значение **fd.** Если **fd** больше нуля, то считается, что откытие файла прошло успешно, а значение **fd** называется ***дескриптором файла.*** Если **fd** равно -1, это означает что запрос на открытие файла не выполнен (файл с указанным именем не существует или может быть недоступен для данного пользователя и т.д.). Следует отметить, что нумерация дескрипторов это атрибут процесса.

Кроме возможности открытия существующего файла имеется функция, обращение к которой позволяет создать новый файл с заданным именем:

int creat(char *name, int perms).

Параметрами данной функции служат:

- **name** - указатель на строку, содержащую полное имя открываемого файла;
- **perms-** значение, специфицирующее права доступа создаваемого файла.

Данный вызов возвращает дескриптор файла, если файл с указанным именем можно создать и -1 в противном случае.

Завершение работы с файлом должно заканчиваться обращение к функции:

close(fd).

Обращение к данной функции обеспечивает “закрытие” доступа к содержимому файла, ассоциированного в данном процессе с дескриптором **fd.**

1.6.3 Чтение/запись данных.

Чтение данных из файла осуществляется при помощи функции:
nr=read(int fd, char * buf, int count)
При обращении к функции **count** байтов из файла с дескриптором **fd** будут помещены в область памяти, на которую указывает указатель **buf**.

Запись в файл осуществляется при использовании функции:
nw=write(int fd, char *buf, int count).
count литер из области памяти на которую указывает **buf** будет записано в файл с дескриптором **fd**.

Рассмотрим системный подход к организации низкоуровнего обмена Рис11. Для этого определим наиболее характерные наборы данных используемые в операционной системе для целей организации низкоуровнего обмена. Эти данные можно подразделить на два класса. Первый - данные ассоциированные с операционной системой, они отражают ситуацию взаимодействия со всеми открытыми к данному моменту времени файлами. Второй - данные ассоциированные с каждым процессом, который их обрабатывает в данный момент времени . Эти данные отражают состав и состояние взаимодействия всех файлов открытых в каждом отдельном процессе.

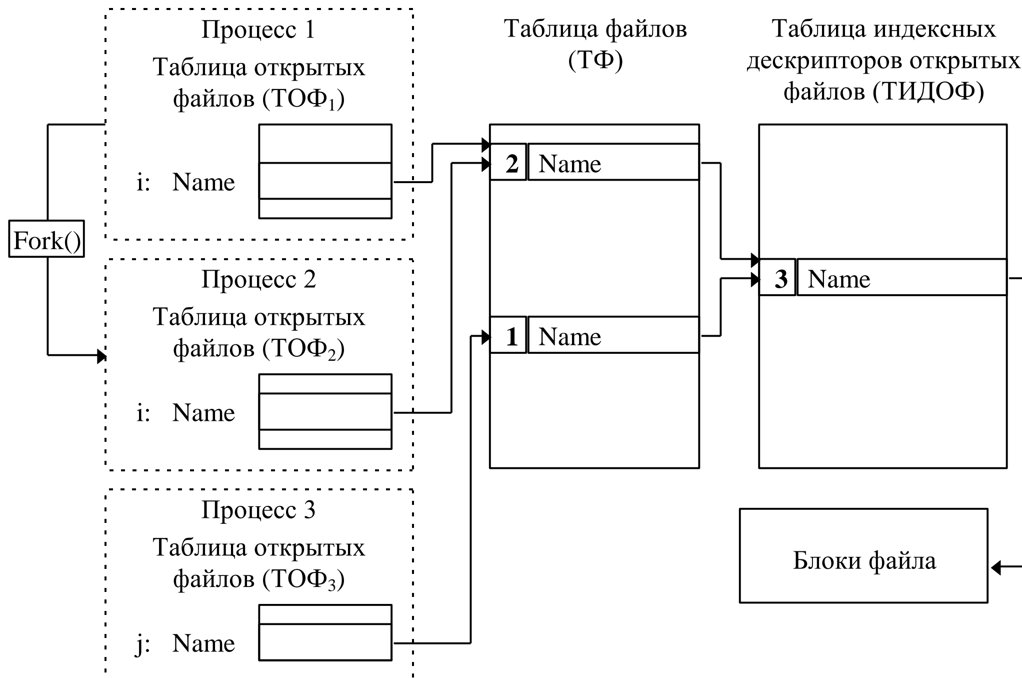


Рис 12. Логическая схема организации обмена.

1.6.4. Данные ассоциированные с операционной системой.

Таблица индексных дескрипторов открытых файлов (ТИДОФ). Это таблица операционной системы, размещенная в оперативной памяти и

содержащая номера и копии индексных дескрипторов всех открытых к данному моменту времени файлов. Кроме того, каждая запись содержит количество ссылок на нее из таблицы файлов(смотрите чуть ниже). Соответственно, все оперативные изменения содержимого индексного дескриптора хранятся в **ТИДОФ**. Полная фиксация всех изменений индексного дескриптора происходит при закрытии файла (осуществляется запись в соответствующее место файловой системы).

Таблица файлов (ТФ). Каждая строка данной таблицы соответствует открытому в каком-то процессе файлу, таким образом количество строк в таблице соответствует числу открытых во всей системе файлов. Это соответствие является взаимнооднозначным, за исключением случаев открытия доступа к файлам унаследованным от родительских процессов (об этом подробнее будет рассказано ниже). Каждая строка этой таблицы содержит информацию о режимах открытия файла, о положении указателя чтения/записи, ссылку на строку **ТИДОФ** в которой размещена копия индексного дескриптора данного файла, а также счетчик “наследственности” - число процессов в которых этот файл передан посредством обращения к функции **fork()** (подробнее об этом будет рассказано ниже).

1.6.5. Данные ассоциированные с процессом.

На уровне данных, ассоциируемых с процессом пользователя, имеется ***таблица открытых файлов процесса(ТОФ)***. Номер строки таблицы - номер файлового дескриптора. Каждая строка таблицы имеет ссылку на соответствующую строку **ТФ.** Следует отметить, что при порождении процесса, интерпретатор команд, который его порождает, предопределяет первые три дескриптора:
0 - стандартный файл ввода информации;
1 - стандартный файл вывод информации;
2 - стандартный файл вывод диагностик.

Рассмотрим, схематично, последовательность действий происходящих с содержимым вышерассмотренных таблиц при выполнении некоторых системных вызовов.

Обращение к функции **fork()** - формирование сыновнего процесса. Как известно, после обращения к данной функции образуется копия исходного процесса. При этом система дублирует таблицу открытых файлов родителя в аналогичную таблицу сына, а также корректирует (увеличивает на единицу) счетчики наследственности в записях таблицы файлов соответствующих файлам унаследованным сыном. Последнее приводит к тому, что и сын и родитель будут использовать один и тот же указатель чтения/записи в наследуемых файлах.

При выполнении функции **open** (для простоты изложения считаем, что запрос на открытие файла корректный: файл с указанным именем существует и нет конфликта с несоответствием прав доступа)**:**

1. по полному имени файла определяется каталог в котором он размещен;
2. определяется номер индексного дескриптора файла;

3. по номеру индексного дескриптора осуществляется поиск в **ТИДОФ**, если запись с заданным номером обнаружена, фиксируем номер соответствующей строки **ТИДОФ** и переходим к пункту 5**;**
4. происходит формирование новой строки соответствующей “новому” индексному дескриптору и фиксируется ее номер;
5. корректируем счетчик ссылок на запись таблицы **ТИДОФ** из таблицы файлов, формируем новую запись в таблице файлов, а также соответствующую запись в таблице открытых файлов процесса, определяется номер этой записи (номер строки) и он возвращается в процесс в качестве файлового дескриптора.

1.6.6. Взаимодействие с устройствами.

Как рассматривалось выше, система **Unix** подразделяет все логические устройства на два класса: байт-ориентированные устройства; блок-ориентированные устройства. Это подразделение во многом зависит от свойств реальных физических устройств и в подавляющем большинстве блок-ориентированное физическое устройство может быть таким же и логическим устройством (аналогично и для байт-ориентированных устройств). Вернемся к рассмотрению Рис.9. Взаимодействие с байт-ориентированными устройствами не представляет особого интереса, поэтому рассмотрим специфические свойства организации обмена с блок-ориентированными устройствами.

Основной особенностью организации работы с блок-ориентированными устройствами является возможность использования буферизации при обменах с реальными физическими устройствами. Суть организации буферизации заключается в использовании пула буферов (размер каждого буфера один блок), каждый из которых может быть ассоциирован с драйвером одного из физических устройств. Пул буферов размещается в оперативной памяти. Далее, выполнение заказа на обмен может происходить по следующим схемам.

<u>Чтение блока.</u> Будем считать что поступил заказ на чтение N-го бока с устройства с номером М.

1. Среди буферов буферного пула осуществляется поиск заданного блока. В случае если обнаружен буфер, содержащий N-й блок устройства М, то фиксируем его номер. Следует отметить, что в этом случае реальное чтение информации с физического устройства не происходит, а операцией чтения является предоставление информации из найденного буфера. Переход на пункт 4.
2. Если поиск заданного буфера неудачен, то осуществляется выбор буфера для чтения и размещения содержимого заданного блока. Если есть свободный буфер (реально это редкая ситуация которая может возникнуть в начальные моменты работы системы) то фиксируется его номер. Переходим к пункту 3. В противном случае, выбираем буфер к которому не было обращений самое продолжительное время. В случае, если в данном буфере установлен признак записи (т.е. буфер использовался ранее для буферизации записи) происходит реальная запись размещенного в буфере блока на физическое устройство, а затем фиксируем его номер и переходим к пункту 3.
3. Осуществляется чтение содержимого N-го блока устройства М, в буфер с зафиксированным ранее номером.

4. Происходит обнуление в данном буфере счетчика времени. Увеличиваем на единицу счетчики времени во всех оставшихся буферах пула. Передаем в качестве результата чтения содержимое зафиксированного буфера. Завершение операции.

Запись блока. Запись блока осуществляется аналогичным образом. Добавляется установка признака записи.

Таки образом организована буферизация при выполнении обменов с блок-ориентированными устройствами. В реальности подобное решение существенно оптимизирует работу **ОС**. Однако у подобного подхода имеют место быть свои недостатки. Система становится критична к несанкционированным выключениям, т.е. ситуациям когда содержимое буферов, имеющих установленные признаки записи еще не записаны на соответствующие устройства, а система прекратила функционирование. Подобные ситуации могут возникать при сбоях в электропитании, отказах аппаратуры или при проявлении программных ошибок в **ОС** и т.п. Другим недостатком использования буферизации является, по сути проявление ее достоинства, т.е. факта разнесения во времени заказа на обмен и реального обмена. К примеру, это может привести к тому что был заказ на обмен со сбоящим физическим блоком устройства для обращающегося процесса будет диагностирован как успешный (т.к. за счет буферизации реального обмена может не быть), а на самом деле, когда очередь дойдет до реального обмена, обмен не произойдет. Но тем не менее эти недостатки заведомо компенсируются преимуществами минимизации числа реальных обменов.

1.6.7. Стандартная библиотека ввода/вывода.

В дополнение к системным вызовам обмена, внешне, аналогичные средства предоставляет системная библиотека обмена. К данной библиотеке относятся традиционные средства ввода/вывода информации, требующие использования стандартного файла заголовков **STDIO.H** (функции **fread(…), fprint(…), fopen(…)).** Рассмотрим, кратко, суть организации данной библиотеки и специфику использования функций ввода/вывода.

1.7.Монтируемость файловой системы.

Корневой каталог **ФС** всегда расположен на системном устройстве, однако это не означает, что и все остальные файлы расположены только на нем. Для связи иерархий файлов, расположенных на разных носителях, применяется монтирование **ФС**, выполняемое системным вызовом

mount(char *special, *name, rwflag)

special - указатель на имя специального файла;

name **-** имя файла, который становится корнем монтируемой **ФС**;

rwflag **-** флаг доступа (0 - разрешена защита, 1 - разрешена запись).

Данный вызов позволяет объявить системе, что **ФС** на сменном носителе **special** доступна при обращении к каталогу **name** , который становится корнем **ФС** монтированного носителя. Т.е. в корневой **ФС** выбирается некоторый существующий файл **,** который после выполнения

mount становится корневым каталогом **ФС**, расположенным на сменном носителе. Через этот каталог такая монтированная **ФС** подсоединяется как поддерево к общему дереву. При этом логически нет разницы между основной, расположенной на системном устройстве, и монтированной файловыми системами.

Однако, между основной и монтированной файловыми системными существует одно отличие, а именно нельзя устанавливать связи между каталогам в одной и файлом в другой **ФС**. Это связано, главным образом, с тем, что при демонтировании **ФС**, трудно было бы отслеживать установленные связи. Т.е. возможность нарушения иерархичности в отдельно взятой **ФС** не распространяется на примонтированную часть. Соответствующий системный вызов для разрыва связи имеет вид

unmount(char *special),

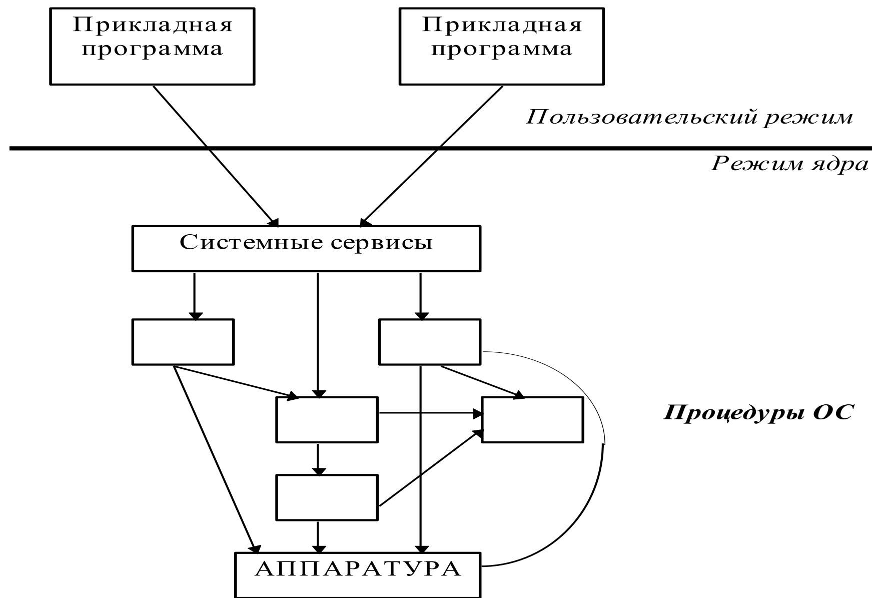
где **special** - указатель на имя специального файла, содержащего монтированную **ФС.**

2. Файловая система Windows NT.

2.1. Особенности ОС Windows NT, влияющие на организацию ввода-вывода

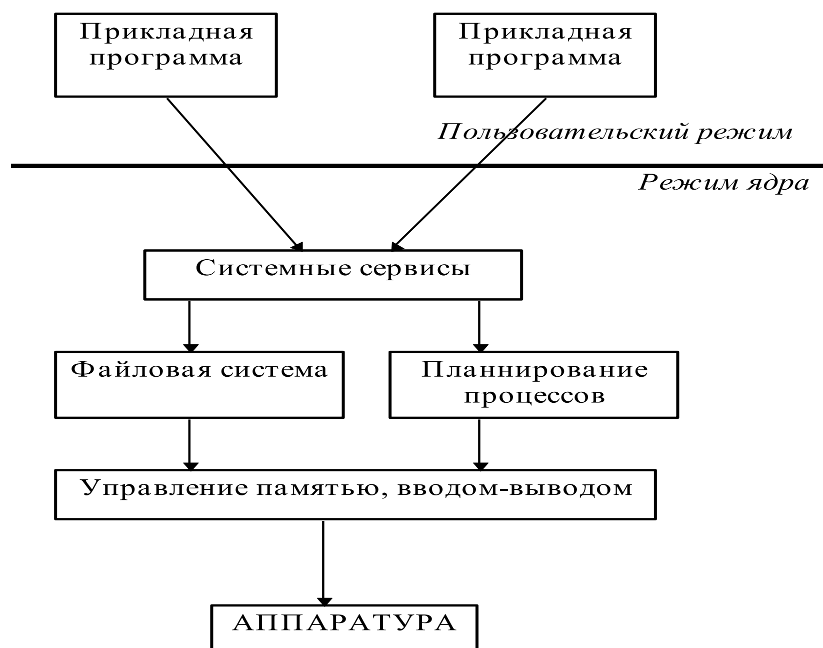
Рассмотрим кратко сособенности архитектуры ОС Windows NT, непосредственно связанные с организацией файловой системы и ввода-вывода. Структура Windows NT базируется на комбинации трех моделей: *клиент-сервер*, *объектная модель* и модели *симметричной мультипроцессорной обработки*. Последняя модель в основном имеет отношение к организации ядра и процессов Windows NT, поэтому здесь мы не будем ее рассматривать, а кратко опишем первые две модели.

Существует множество способов структурной организации ОС. Среди них можно выделить три основных - это *монолитный*, *послойный* и *клиент-сервер*. Монолитной структурой обладают как правило простые и небольшие по размеру ОС, например MS-DOS. Т.е. ОС организована как набор процедур, причем каждую из них может вызвать пользовательская программа. Для большинства монолитных ОС, за исключением наиболее простых существует разделение выполняемого кода на код непосредственно ОС, исполняемого в привилигированном режиме (далее *режим ядра*), и код пользовательских программ, исполняемым в *пользовательском режиме*. Данный режим отличается ограниченным доступом к системным ресурсам, данным и ограниченным набором интерфейсов, т.е. непосредственно действий, которые может выполнить программа.



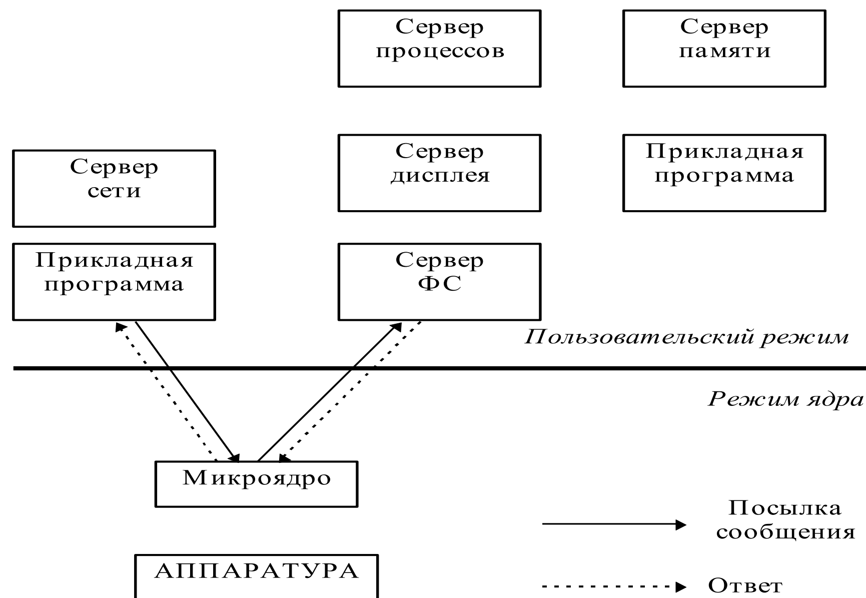
Когда программа пользовательского режима обращается к функциям монолитной ОС (далее будем говорить, что в этом случае происходит вызов *системного сервиса*), то процессор перехватывает данный вызов и переводит вызывающий процесс в режим ядра, до завершения выполнения системного вызова. После его окончания продолжается выполнение пользовательской программы в обычном режиме. Пример данной модели функционирования - вызов функии MS-DOS через прерывание 21h.

Другой, более современный подход к структурированию ОС, подразумевает разбиение кода ОС на модули, наслоенные один поверх другого. При этом каждый модуль предоставляет набор функций (интерфейсов), и эти функции доступны только соседним по иерархии модулям.



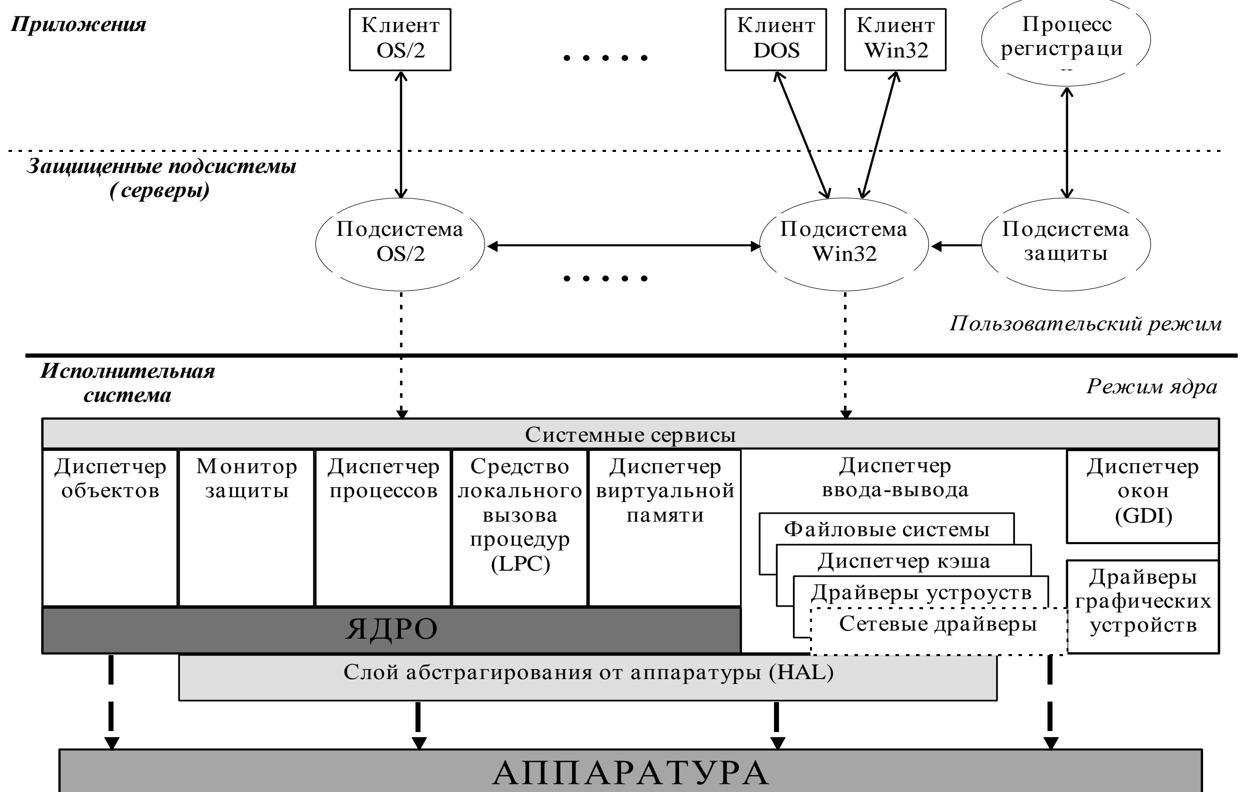
Основным преимуществом послойной ОС является то, что код каждого слоя доступен только соседним слоям и только через определенные интерфейсы, что повышает надежность ОС и позволяет например заменить один из уровней ОС, что невозможно сделать в монолитных ОС.

Третий, наиболее прогрессивный подход к структурированию ОС - это модель клиент-сервер. Основная идея этой модели состоит в том, чтобы разделить ОС на несколько процессов, каждый из которых реализует определеннный набор сервисов, как то распределение памяти, контроль доступа и т.п. Каждый *сервер* выполняется в пользовательском режиме и проверяет не обратился ли к нему за обслуживанием какой-либо клиент. *Клиентом* может быть другой серверный процесс или пользовательский процесс, посылающий серверу сообщение с запросом на обслуживание, передача сообщений и все межпроцессное взаимодействие организуется через ядро (микроядро), код которого исполняется в привилигированном режиме. При таком подходе ОС обладает свойствами большей надежности, расширяемости, гибкости, возможностью замены компонентов ОС «на ходу» и, что наиболее важно, возможностью использования на распределенных вычислительных системах. Ниже на рисунке приведен пример идеальной клиент-серверной ОС, на самом же деле многие серверные процессы приходится исполнять в режиме ядра, например планирование процессов, работу с виртуальной памятью, использование драйверов устройств и т.д. Наиболее соответствующей требованием клиент-серверной ОС на данный момент является ОС Mach, архитектура которой, по утверждению разработчиков Windows NT, во многом повлияла на архитектуру самой NT.



Можно утверждать, что структура Windows NT, обладает свойствами как клиент-серверной так и послойной модели. Рассмотрим подробнее основные компоненты и способы их взаимодействия.

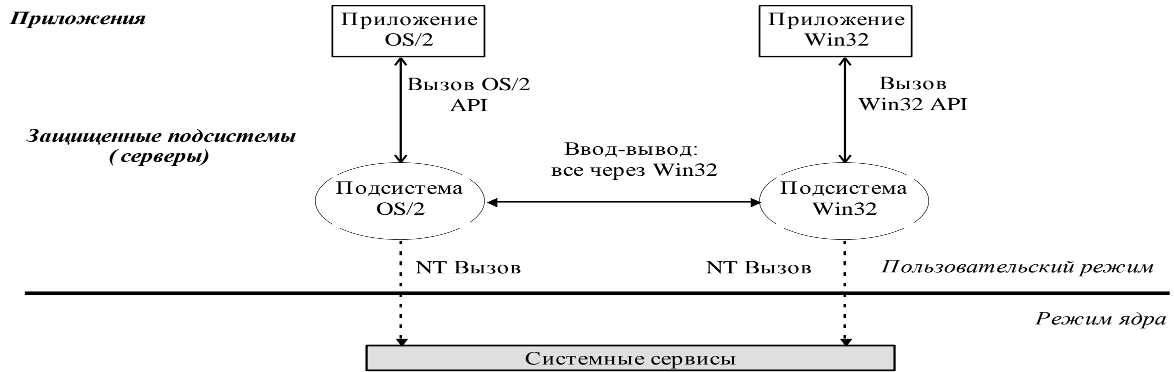
Структура ОС Windows NT



Структурно Windows NT можно разделить на две части: работающую в пользовательском режим - *защищенные подсистемы*, и работающую в режиме ядра - *исполнительная система*.

Защищенные подсистемы.

Каждая защищенная подсистема - это серверный процесс (как правило состоящий из нескольких нитий). Термин «сервер» подразумевает, что каждая защищенная подсистема предоставляет своим клиентам некоторый набор интерфейсов (API), который используют другие программы. Когда приложение вызывает одну из процедур API данного сервера, то этому серверу посылается сообщение при помощи механизма LPC - *локального вызова процедур* (это специально оптимизированный механизм для локальной передачи сообщений). Сервер же выполняет некоторую работу и посылает ответ вызвавшему его клиенту. В Windows NT существует два типа защищенных подсистем: *подсистемы среды* и *интегрированные подсистемы*. Подсистема среды это сервер пользовательского режима, реализующий API некоторой ОС. Наиболее важная среди них - подсистема Win32, которая предоставляет прикладным программам API 32-разрядной Windows. Помимо этого Win32 отвечает за графический интерфейс пользователя и за «консольный» ввод-вывод всех приложений. Это значит, что остальные подсистемы среды такие как DOS, OS/2, 16-битный Windows для обеспеения этих функций обращаются к подсистеме Win32. Но для обеспечения остальных функций, таких как например файловый ввод-вывод, подсистемы среды работают с системными сервисами напрямую. Например, если 16-битное OS/2 приложение вызывает функцию открытия файла, то защищенная подсистема среды OS/2 преобразует, этот вызов в NT вызов открытия файла, получает NT описатель открытого файла, преобразует его в OS/2 формат и предоставляет вызвавшему приложению.



Интегрированные защищенные подсистемы. Набор данных защищенных подсистем менялся в от версии к версии. На данный момент он включает в себя сетевые сервисы рабочей станции и сервера, подсистему защиты и ряд других сервисов. Наиболее важной интегрированной защищенной подсистемой является подсистема защиты. Основная ее задача это регистрация и отслеживание правил контроля доступа для данного локального компьютера. Она отслеживает какие пользователи имеют какие привелегии, какие системные ресурсы подлежат аудиту и т.д. Помимо этого она ведет базу данных учетных записей пользователей, содержащую пользовательские идентификаторы, пароли, права доступа, информацию о пользовательских группах и т.д. Процесс регистрации и идентификации пользователя при входе в систему также проходит через подсистему защиты. А именно:

1. *Процесс регистрации* (logon process) ожидает ввода от пользователя. Одновременно может быть активно несколько таких процессов, каждый работает со своим классом устройств, например, первый через консоль, а второй через сетевое соединение. Обнаружив пользователя, желающего войти в систему, данный процесс запрашивает у него идентификатор и пароль, после чего посылает эту информацию подсистеме защиты.
2. Подсистема защиты проверяет правильность введенной информации. И в случае корректного ввода генерирует объект, называемый *маркером доступа*, который будет идентифицировать данного пользователя и его права на протяжении всей сессии. Подробнее про контроль доступа к ресурсам системы будет рассказано ниже.
3. В случае входа пользователя не по сети, а с локальной консоли, подсистема защиты создает desktop - процесс вошедшего пользователяи передает его системе Win32, которая его запускает, для предоставления пользователю графического интерфейса. Для Windows 4.0 - этот процесс Explorer, для более ранних версий это Program Manager.

Исполнительная подсистема.

Как говорилось выше исполнительная система - это часть Windows NT, выполняющаяся в режиме ядра. Она состоит из ряда компонент, каждая из которых реализует два набора функций: *системные сервисы*, к которым могут обращаться как защищенные подсистемы так и другие компоненты исполнительной части; а также *внутренние процедуры*, которые могут использовать только другие компоненты исполнительной подсистемы. Хотя исполнительная подсистема и предоставляет системные сервисы, сходные с API подсистем среды, она принципиально отличается от остальной части ОС. В

то время как все серверы защищенной подсистемы выполняются постоянно, т.е. соответствующие процессы всегда активны, то исполнительная система не работает постоянно, а активизируется, когда происходит какое-либо важное системное событие. Например, при вызове системного сервиса или при прерывании от внешнего устройства. Компоненты исполнительной системы поддерживают независимость друг от друга. Структуры системных данных могут использоваться только одним из компонент исполнительной части, а все взаимодействие происходит только через интерфейсы. Это означает, что если правильно реализовать весь набор интерфейсов и системных сервисов некоторой компоненты, то в системе ее можно заменить на свою собственную. Т.е. появляется возможность покомпонентой замены ОС. Перечислем компоненты исполнительной части и кратко опишем их основные функции (см. рисунок).

- *Диспетчер объектов*. Создает, поддерживает и уничтожает объекты исполнительной части. Подробнее рассматривается ниже.
- *Монитор защиты.* Обеспечивает защиту ресурсов и объектов ОС.
- *Диспетчер процессов*. Создает, завершает, приостанавливает и запускает процессы и нити ОС.
- *Средство локального вызова процедур*. Передает сообщения между клиентскими и серверными процессами в пределах компьютера. Является вариантом RPC (remote procedure call), специально оптимизированным для работы в рамках одной машины.
- *Диспетчер виртуальной памяти*. Реализует управление виртуальной памятью машины.
- ***Ядро.*** Реагирует на прерывания и исключения, осуществляет переключение выполняющихся нитей, межпроцессорную синхронизацию. Предоставляет набор элементарных объектов и интерфейсов, которые используются для построения более сложных объектов исполнительной части.
- ***Система ввода-вывода.*** Она состоит из следующих подсистем.
    1. *Диспетчер ввода-вывода*. Реализует средства ввода-вывода, не зависящие от типа устройства и устанавливает модель ввода-вывода для исполнительной части NT.
    2. *Файловые системы*. Драйверы NT, принимающие запросы файлового ввода-вывода и транслирующие эти запросы в запросы для драйверов конкретных устройств.
    3. *Сетевой редиректор и сетевой сервер*. Драйверы файловой системы, передающие запросы удаленного ввода-вывода на другие машины в сети и принимающие от них информацию.
    4. *Драйверы устройств*. Низкоуровненвые драйверы, напрямую работающие с аппаратурой.
    5. *Диспетчер кэша*. Обеспечивает запись на устройства в фоновом режиме и кэширование читаемой информации в виртуальной памяти.
- *Слой абстрагирования от аппаратуры*. Помещает кодовую прослойку между исполнительной системой и аппаратной платформой, скрывая аппаратно зависимые детали, такие как интерфейсы ввода-вывода, контроллеры прерываний, механизмы межпроцессорных связей. Данный уровень обеспечивает большую переносимость Windows NT, хотя все равно для переноса на другие платформы требуется переписывать значительную часть ядра и менеджера виртуальной памяти.

В заключение данной части можно сказать, что работа защищенных подсистем, их взаимодействие с исполнительной системой и клиентскими приложениями осуществляется на принципах клиент-сервер, организация ввода-вывода в исполнительной части, часть ядра и уровень абстагирования от аппаратуры являются «послойными» элементами. Внутри же исполнительной части ее компоненты взаимодействуют также через строго определенные интерфейсы, хотя их структура не является «послойной».

2.1.1. Объектная модель и контроль доступа в Windows NT.

Сначала дадим определение объекта исполнительной части Windows NT.

Объект - это статический экземпляр определенного в системе *типа* (*класса*) *объектов,* существующий во время работы системы.

Тип объекта включает определенный системой тип данных, *сервисы* (*методы*) работающие с образцами этого типа данных и набор *атрибутов* объекта.

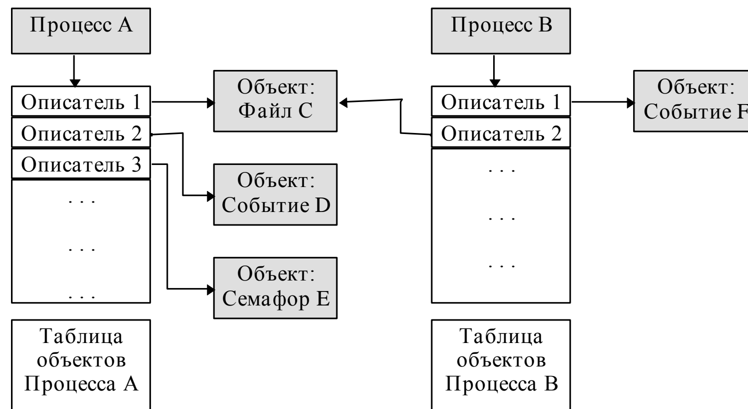
Атрибут объекта - это поле данных внутри объекта, частично определяющее его состояние.

Сервис (*метод*) *объекта* - способ манипулирования объектом, обычно производит некоторые действия над атрибутами. Основная особенность объектов исполнительной системы Windows NT это то, что внутренняя структура и организация объекта всегда скрыта и все атрибуты доступны только через использование соответствующих сервисов объекта. Несмотря на использование понятия «объект» Windows NT не является объектно-ориентированной ОС. Основные цели, которых хотели достичь разработчики Windows NT, реализуя объекты были следующие:

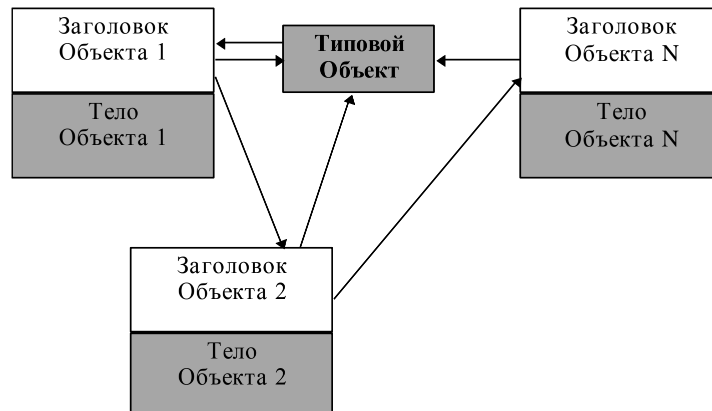
- Обеспечить общий унифицированный механизм работы с системными ресурсами
- Сделать единые средства защиты и контроля для всей системы
- Обеспечить единую удобную схему именования элементов и ресурсов системы
- Облегчить поддержку различных сред ОС, что требует частого преобразования представления одной и той же сущности (например описателя файла) в различные типы (например в формат OS/2 из NT и обратно)

Структура объектов.

Как и в большинстве ОС основной единицей работы в NT является процесс. Для того чотбы процессу использовать некоторый ресурс системы ему необходимо запросить описатель объекта соответствующего ресурса. Описатели объектов уникальны в то время как сами объекты могут совместно использоваться несколькими процессами. Информация об используемых процесом объектах хранится в специальной таблице объектов процесса.



Описатель помимо ссылки на объект и идентификатора предоставленного доступа хранит информацию о режиме наследование. Т.е. получат ли процессы созданные данным процессом копию данного описателя и с какими правами доступа. В NT существует понятие «типового объекта». Такой объект используется при порождении объектов данного типа и содержит атрибуты общие для фсех объектов данного типа. Например, объект «процесс регистрации» имеет типовой объект «процесс» и т.д.



Помимо этого все объекты, связанные с одним типовым, объединены в список, что позволяет системе быстро просматривать все объекты данного типа. Сам объект состоит из заголовка и тела. Для поиска и хранения имен объектов используется специальная структура, называемая «каталогом объектов». Она похожа на структуру ФС в ОС Unix и позволяет быстро находить объект по его имени и типу. Подробное описание данной структуры заняло бы достаточно много времени и мы не будем рассматривать данную структуру в деталях, а приведем простой пример, поясняющий ею сущность. Многие может быть видели сообщения об ошибке типа: «Ошибка записи на \Device\HardDisk0\Partition3», в этом сообщении «\Device\HardDisk0\» есть путь в каталоге объектов до объекта «Partition3».

Обязательные атрибуты заголовка объекта

Атрибут	Назначение
Имя объекта	Делает объект видимым для других процессов
Каталог объектов	Ссылка на иерархическую структуру, где хранятся имена объектов

Дескриптор защиты	Определяет кто и как может использовать данный объект
Расход квоты	Задает квоту на использование ресурсов, которая списывается с процесса, когда тот открывает описатель данного объекта
Счетчик открытых описателей	Подсчитывает количество открытых описателей описателей данного объекта
База данных открытых описателей	Список процессов, работающих с данным обеъектом
Временный/постоянный	Указывает можно ли уничтожать этот объект и освобождать память
Режим: ядра/ полльзовательский	Определяет доступность объекта в пользовательском режиме
Указатель на типовой объект	Ссылается на типовой объект, который содержит атрибуты, общие для набора однотипных объектов

Обязательные объектные сервисы

Сервис	Назначение
Закрыть	Закрывает описатель объекта
Дублировать	Обеспечивает совместное использование объекта путем дублирования его описателя и передачи его другому процессу
Опросить объект	Получает информацию о стандартных атрибутах объекта
Опросить защиту	Получает дескриптор защиты
Установить защиту	Изменяет дескриптор защиты
Ждать одного объекта	Используется для синхронизации
Ждать нескольких объектов	Используется для синхронизации

Атрибуты типового объекта

Атрибут	Назначение
Имя типа объекта	Название объекта данного типа (например «процесс», «событие» и т.д.)
Типы доступа	Тип доступа, который может быть запрошен процессом при открытии
Возможности синхронизации	Может ли процесс синхронизироваться, использую объект данного типа
Резидентный/ нерезидентный	Указывает возможно ли выгружать объекты данного типа из оперативной памяти
Методы	Набор процедур, которые могут вызываться Диспетчером Объектов в определенные моменты (аналог тригеров в БД).

Все операции пользовательской программы над объектом, возможны только через описатель, а поиск объекта по описателю, создание объектов и выделение новых описателей уже существующих объектов - это задачи диспетчера объектов. Соответственно он осуществляет:

- Выделение памяти при создании объекта
- Определение дескриптора защиты вновь созданного объекта
- Создание и поддержание структуры каталога объектов
- Создание описателя для запрашивающего процесса
- Защиту объекта. При запросе описателя диспетчер объектов вызывает *монитор защиты*, который проверяет соответствие прав доступа вызывающего процесса дескриптору защиты данного объекта. Если процесс имеет соответствующие права, то *монитор защиты* формирует список прав для вызывающего процесса, которые диспетчер объектов помещает в описатель. И при каждом последующем обращении процесса к сервису данного объекта, проверяется возможность вызова данного объектного сервиса вызывающим процессом.
- Контроль использования объекта. Имеется в виду проверка на возможность удаления временного объекта. Временный объект удаляется из памяти, если счетчик открытых описателей равен нулю, т.е. его никто не использует и можно освобождать память.
- Контроль использования ресурсов процессом. При открытии нового описателя для процесса проверяется соответствие используемых ресурсов установленным для него границам (квотам). В списке обязательных атрибутов объекта есть поле «квота», которое показывает насколько увеличится используемая процессом память если он получит описатель и будет использовать данный объект. Если суммарное значение для процесса больше, чем граница, заданная администратором для процессов, созданных данным владельцем, то процесс дескриптора не получит.
- Вызов методов типового объекта. Чтобы унифицировать работу со всеми типами объектов, включая объекты, которые может создавать и добавлять в систему пользователь, в Диспетчере Объектов реализован механизм, позволяющий в определенных ситуациях вызывать заданные методы определенного типового объекта. Например, метод «открыть» вызывается при открытии описателя объекта данного типа. Эти методы определены в типовом объекте (см. выше) и одинаковые для всех объектов такого типа. Их

можно сравнить с конструкторами / деструкторами языка C++ или тригерами в БД.

Защита объектов.

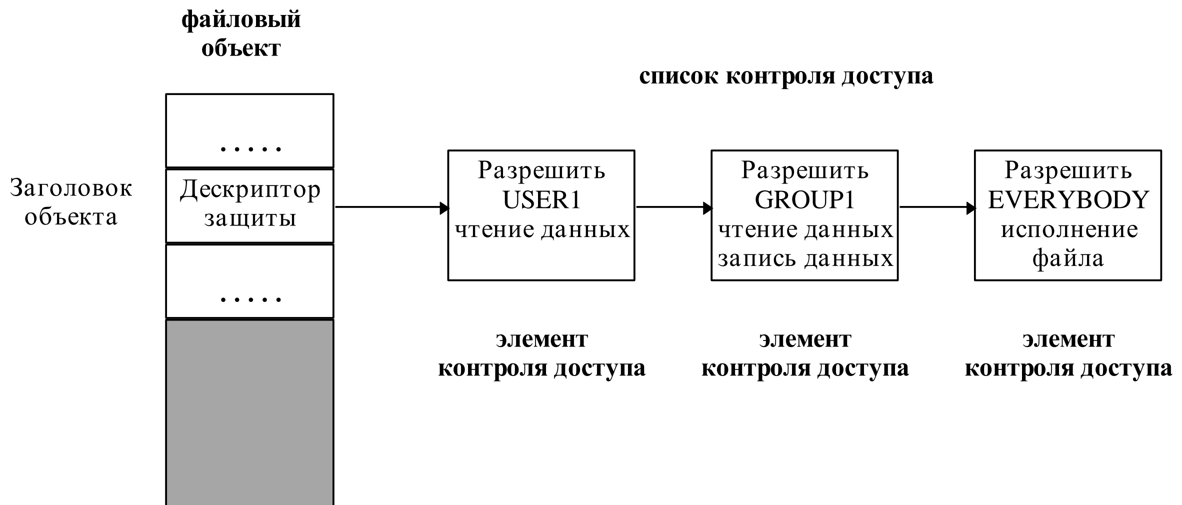
Как говорилось выше, каждый процесс связан со специальным объектом, называемым *маркером доступа*. Первоначально маркер доступа генерируется системой при регистрации пользователя и дальше наследуется (может с ограничениями) всеми процессами, которые вызывал пользователь или, запущенные им приложения.

Атрибуты маркера доступа

Атрибут	Назначение
Идентификатор защиты	Имя пользователя или группы
Идентификаторы групп	Список групп к которым он принадлежит
Привилегии	Список личных привилегий пользователя
Владелец по умолчанию	Имя владельца процесса (как правило совпадает с именем пользователя)
Первичная группа	Основная группа пользователя
Список контроля доступа (ACL) по умолчанию	Список контроля доступа, который присваевается, создаваемым объектам по умолчанию

Список контроля доступа представляет собой последовательность *элементов контроля доступа*. Каждый элемент содержит идентифиатор защиты и связанные с ним права доступа.

Пример списка контроля доступа (ACL) для файлового объекта



При создании нового объекта список контроля доступа (ACL) определяется следующим образом:

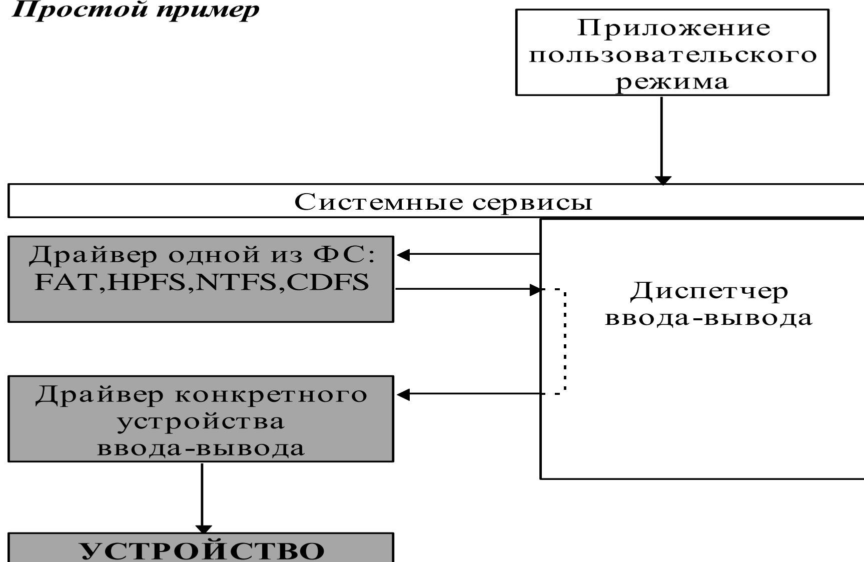
- Если ACL задан явно при создании объекта, то он пишется в дескриптор
- Если ACL не задан явно, но у объекта есть имя, то подсистема защиты ищет ACL в каталоге объектов, где будет сохранен данный объект. Некоторые элементы ACL каталога могут быть помечены как наследуемые и тогда ACL объекта формируется из них.
- Если ни первый ни второй вариант не проходит система берет ACL из маркера доступа создающего процесса.

Алгоритм проверки прав доступа к объекту очень прост. Просматривается весь ACL пока не будет найден разрешающий элемент. Если элемент не найден, то в доступе (или вызове сервиса) вызывающему процессу будет отказано. Параллельно система осуществляет аудит, т.е. фиксирует кем и когда были предприняты попытки вызова сервисов или использования ресурсов, которые не соответствуют правам вызывающего.

2.1.2. Организация ввода-вывода.

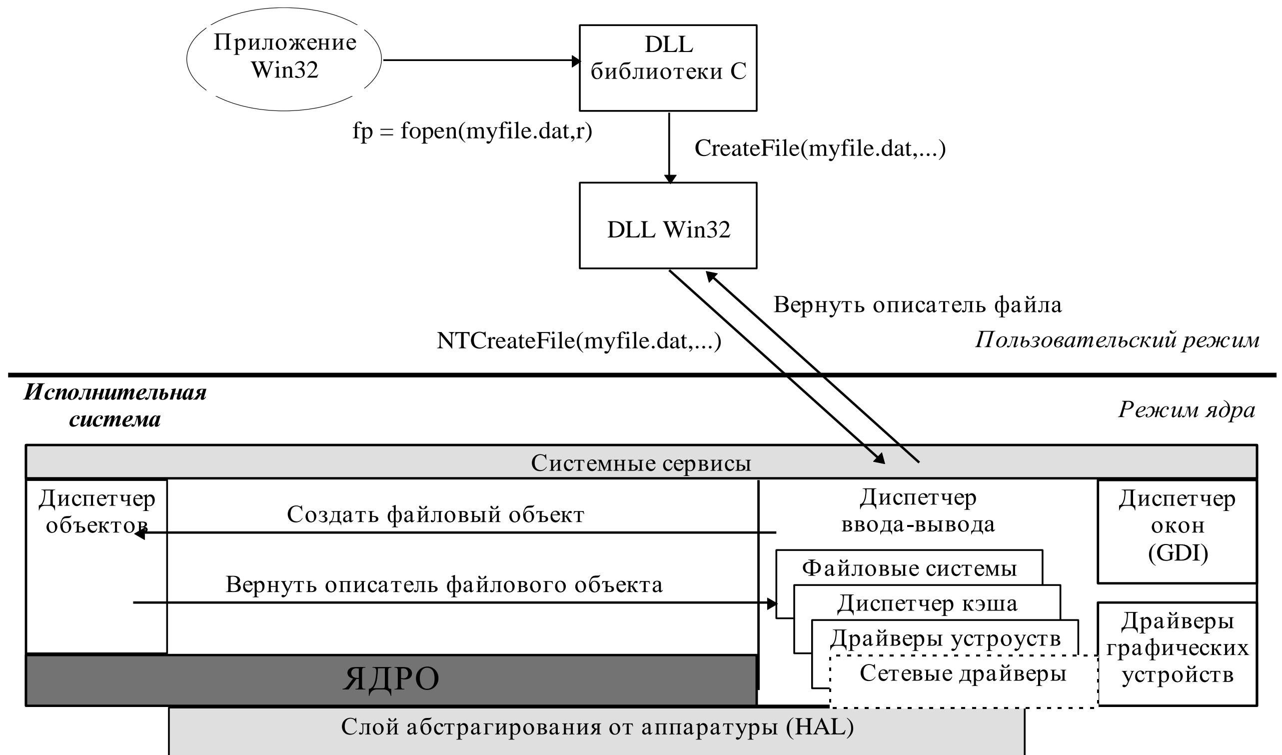
Система ввода-вывода исполнительной системы NT - это часть кода ОС, получающая через свои системные сервисы и внутренние интерфейсы запросы ввода-вывода от приложений пользовательского режима и от других компонент режима ядра и передающая их в преобразованном виде, устройствам ввода-вывода. Система ввода-вывода в NT является классической «послойной» системой. Система ввода-вывода NT называется *управляемой пакетами* (packet driven). Это значит, что каждый запрос ввода-вывода представляется в виде *пакета запроса ввода-вывода* (I/O request packet-IRP) и все функционирование системы ввода-вывода, связанное с этим запросом полностью определяется содержимым этого пакета, которое меняется при переходе пакета от одной компоненты системы ввода-вывода к другой. Т.е. IRP - это структура данных, управляющая обработкой операции ввода-вывода на каждой стадии ее выполнения. Рассмотрим основные компоненты системы ввода-вывода и связи между ними.

Наиболее важная часть - это *Диспетчер ввода-вывода*, он определяет модель функционирования системы. Реально он не управляет вводом-выводом, а лишь создает IRP и осуществляет его маршрутизацию от драйвера к драйверу. Реальную же работу выполняет драйвер. Получая IRP, он выполняет указанную в нем операцию и возвращает его с обновленным содержимым обратно Диспетчеру ввода-вывода или другому драйверу (опять таки через диспетчер ввода-вывода). Понятие «драйвер» в ОС Windows NT более широко, чем в других ОС. В частности под драйвером понимается не только драйвер устройства, но и более сложные драйверы файловых систем, сетевые драйверы, драйверы отказоустойчивых систем и т.д.



В отличие от ОС Unix, где весь ввод-вывод основан на понятии *виртуального файла*, как универсального источника или приемника информации, в ОС Windows NT ввод-вывод основан на понятии *файлового объекта*. Аналогом Unix файлового дескриптора в Windows NT служит *описатель файлового объекта*. Весь контроль доступа, учет ресурсов, аудит и т.д. осуществляется как

и со всеми остальными объектами системы. Рассмотрим, что происходит в системе, когда пользовательская программа на С вызывает функцию **fopen**.



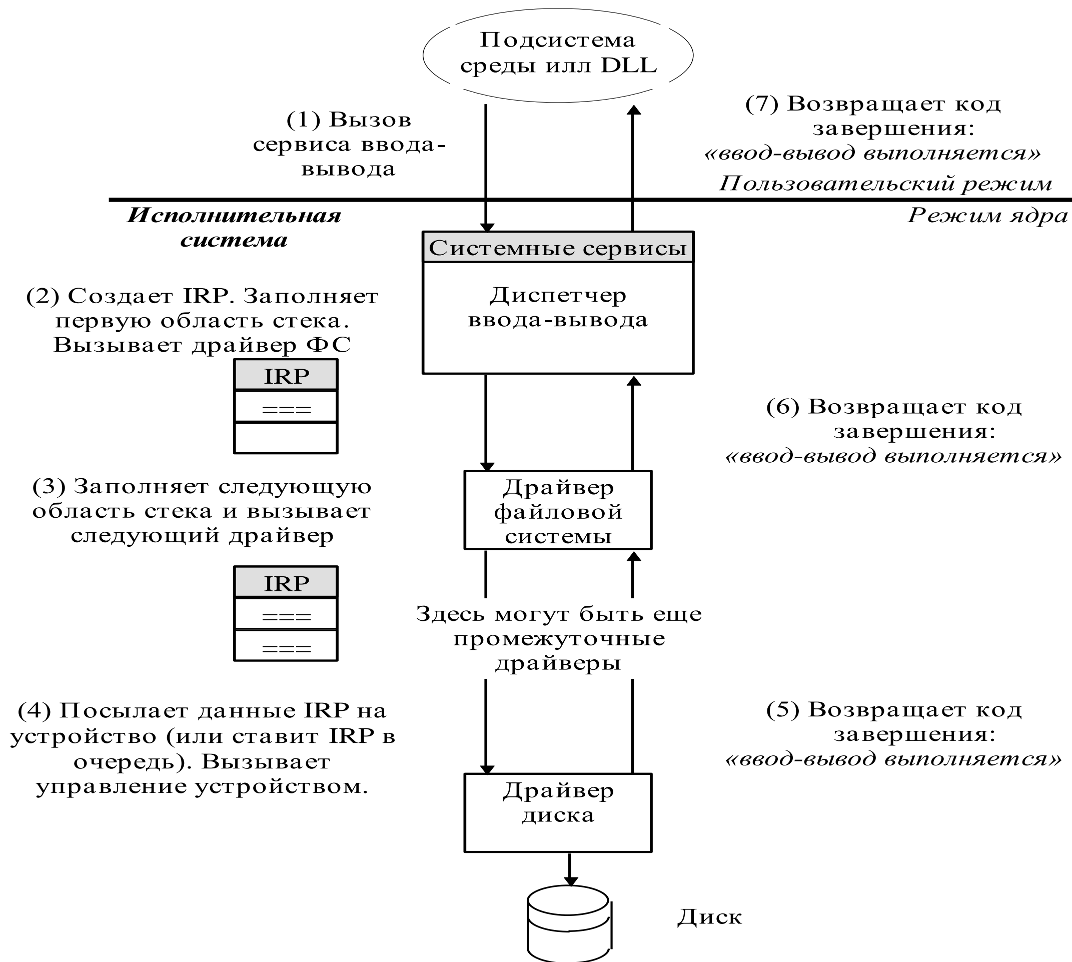
Структура файлового объекта.

Тип объекта	**Файл**
Атрибуты тела объекта	Имя файла Тип устройства Режим разделения Байтовое смещение Режим доступа . . .
Сервисы	Создать файл Открыть файл Читать Писать . . .

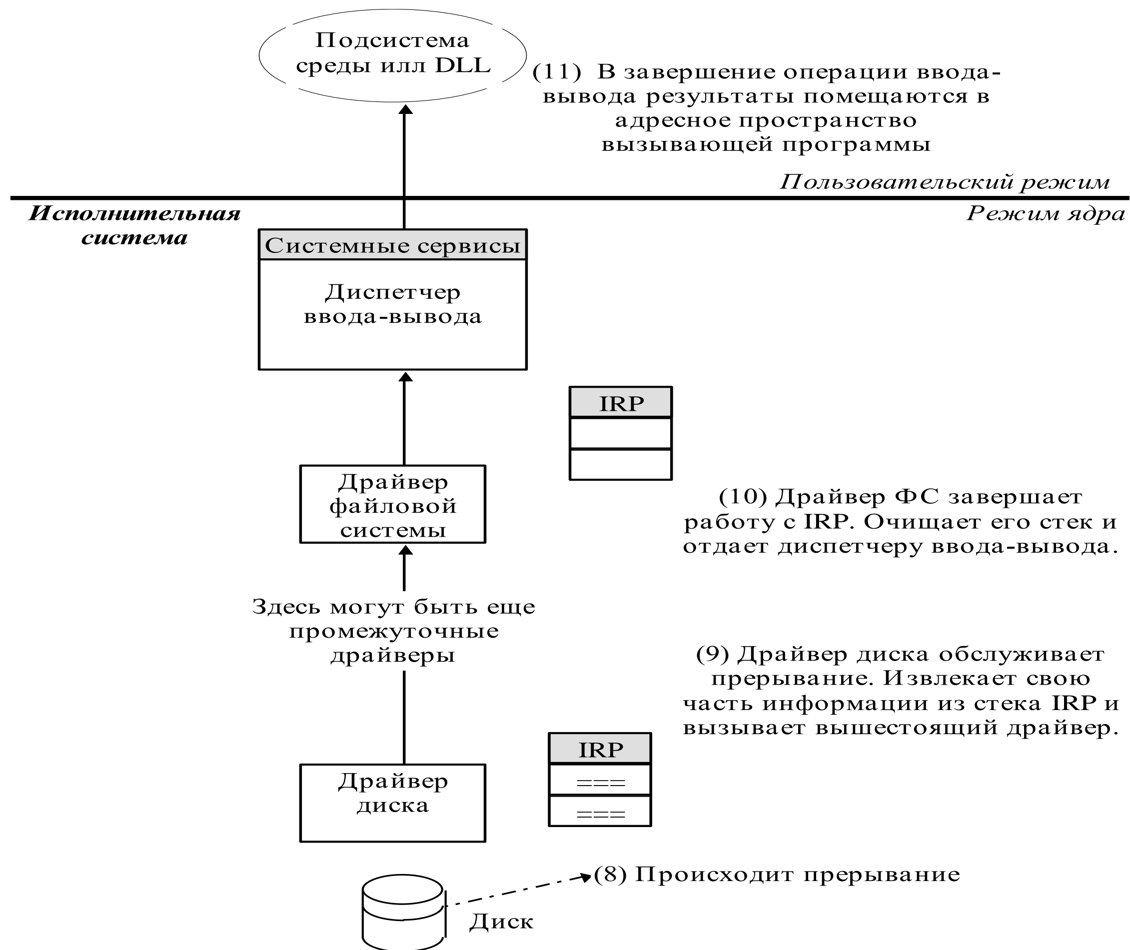
Соответственно, вызов пользовательского приложения например на чтение или запись преобразуется защищенной подсистемой среды в вызов соответствующего сервиса файлового объекта. Следует также заметить, что драйверы, пакет IRP и даже устройства рассматриваются в Windows NT как объекты, но мы не будем подробно разбирать структуру данных объектов. Рассмотрим по-этапно функционирование системы ввода-вывода при обращении к ней какого-либо пользовательского приложения.

Диспетчер ввода-вывода получает запрос через один из своих системных сервисов, создает IRP пакет и передает его драйверу ФС. В зависимости от содержимого пакета драйвер ФС может просто переслать данный IRP нижележащему драйверу (драйверу устройства - простейшем случае или в более сложном варианте: отказоустойчевому драйверу или драйверу тома), а может сгенерировать ряд дополнительных пакетов и также отослать их следующему драйверу. Каждый IRP содержит область, называемую стеком, куда каждый следующий драйвер вносит дополнительную служебную

информацию (необходимые операции с пакетом и параметры данных операций), когда пакет проходит «к устройству» и читает (стирая) информацию из стека, когда пакет возвращается «с устройства».



После окончания записи на диск генерируется прерывание и запрос ввод-вывода завершается по обратной траектории.



Нетрудно видеть, что все операции ввода-вывода в Windows NT ассинхронны, хотя защищенные подсистемы среды могут предоставлять своим приложениям синхронные API ввода-вывода, самостоятельно реализуя ожидания оканчания операции. В данной модели функционирования системы ввода-вывода реализовано большое колличество оптимизирующих алгоритмов и средств. Например, если к драйверу ФС пришел запрос на чтение данных, которые разбросаны по разным участкам диска, то драйвер ФС генерирует набор IRP пакетов, которые обрабатываются независимо и параллельно, причем каждый содержит запрос на чтения одного локального участка данных. По оканчании драйвер ФС собирает их обратно в один пакет и отдает диспетчеру ввода-вывода. Драйверы всех уровней являются динамически подгружаемыми, а значит не занимают ОП, если их никто не использует. Помимо этого, как говорилось выше, все драйверы также являются объектами со строго специфицированными интерфейсами, а значит они легко заменяются.

2.2. Файловая система NTFS.

2.2.1. Основные особенности NTFS и модель функционирования

Кратко по пунктам опишем основные особенности NTFS. Мы будем рассматривать только функции и особенности, реализованные на момент выхода Windows NT 4.0.

1. *Работа с носителями большого объема.*

NTFS распределяет дисковое пространство по кластерам. Размер кластера называется *кластерным множителем*. Физически кластер - это последовательность секторов на диске. Для нумерации кластеров используется 64-битовое число, т.е. число кластеров может быть свыше 16.000.000.000.000.000.000, каждый объемом от 512 байт до 64 Кбайт в зависимости от установок при форматировании и объема диска. В частности нельзя отформатировать диск 1Гбайт с кластером 64Кбайта.

2. *Восстанавливаемость*

В случае программного или аппаратного сбоя и порчи данных на диске NTFS автоматически восстанавливает последнее “правильное” (по терминологии - “согласованное” или консистентное) состояние данных. Для этих целей используется модель обработки транзакций и резервное копирование наиболее критических данных ФС. При обращении к испорченному участку диска NTFS автоматически помечает его как “плохой”, и в случае наличия резервной копии (всегда для системных данных и для пользовательских данных в случае использования отказоустойчивых устройств) перераспределяет дисковое пространство для восстановления испорченного участка в другой части диска.

3. *Избыточность данных и отказоустойчивость*

В общем случае свойство восстанавливаемости NTFS гарантирует только возможность продолжения работы системы после сбоя и сохранность системных данных, но не гарантирует полной сохранности всех пользовательских данных, записанных на диск до сбоя. Для критических приложений, таких как банковские, правительственные, военные и др. системы такая модель неприемлема. В результате в NTFS была добавлена поддержка отказоустойчивых дисковых хранилищ, использующих RAID технологию, причем в связи с многоуровневой организацией драйверов устройств NTFS использование таких устройств так же прозрачно для пользователя и прикладных программ. (Данная возможность есть только в Windows NT Advanced Server).

4. *Защита от несанкционированного доступа.*

Организация контроля доступа в NTFS непосредственно реализована через объектную модель NT. Каждый открытый файл представляется в виде файлового объекта со своим дескриптором защиты, который хранится на диске как часть файла. Прежде чем процесс сможет открыть описатель любого объекта, в том числе и файлового, система контроля проверяет, есть ли у процесса соответствующие права.

5. С*оответствие UNIX стандартам*

NTFS соответствуеют стандартам файловых систем, аналогичных Unix ФС, т.е. например, различие имен в зависимости от регистра букв, отметки времени изменения файла, иерархическую структуру ФС и т.д. Реализована также поддержка жестких связей (hard links). Единственное, что пока не поддерживается из стандарта это символические связи (symbolic links).

7. *Множественные потоки данных*

В NTFS с каждым атрибутом файла таким как имя, дата создания, дескриптор защиты и т.д. связан отдельный поток данных. Поскольку непосредственно

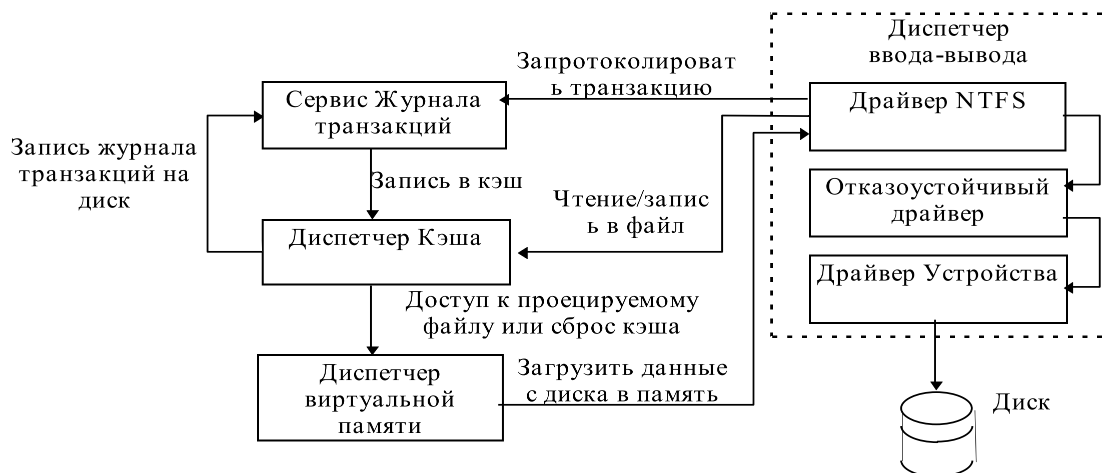
данные в файле интерпретируются как отдельный атрибут и существует возможность добавлять новые атрибуты, то легко добавлять новые потоки данных, связанные с этим файлом. Например file.dat : stream1 и file.dat : stream2 есть два различных потока, открытых для одного файла данных. Каждый поток обладает собственными атрибутами, такими как выделенный ФС размер дискового пространства, реально использованный размер дискового пространства и т.д., поддерживается система захватов (lock), используемых для блокировки частей потока при параллельном доступе, но реально пока используются механизмы захвата только на уровне файла.

8. *Сжатие данных*

NTFS позволяет осуществлять автоматическую архивацию/разархивацию при работе с ФС, но алгоритмы и принципы функционирования данной подсистемы достаточно интересны и сложны, они заслуживают отдельного рассмотрения, поэтому в данной работе мы не будем их рассматривать.

Рассмотрим модель функционирования NTFS.

С точки зрения компонент ФС Windows NT NTFS - это комплект загружаемых драйверов, предназначенных для обработки запросов ввода-вывода (см. выше).



Все драйверы неявно вызываются приложениями, обращающимися к API ввода-вывода. Вызов API передается соответствующему системному сервису, который выбирает необходимые драйверы и вызывает их (в момент обращения драйвер может не находится в оперативной памяти). Все драйверы разных уровней общаются друг с другом через диспетчер ввода-вывода стандартным образом, поэтому загрузка/выгрузка или замена драйвера на другой того же уровня не влияет на работу остальных драйверов. Каждый драйвер взаимодействует только с соседними по вертикали, такая организация ввода-вывода называется послойной и представляет собой многоуровневую модель взаимодействия. Помимо друг друга драйверы взаимодействуют с другими компонентами исполняемой части NT как это показано на рисунке. Эти компоненты - *Сервис журнала транзакций*, обеспечивающий обработку транзакций (подробнее в части “восстанавливаемость”), и *Диспетчер кэш,* организующий кэширование файлов в оперативной памяти с помощью диспетчера виртуальной памяти. Для ускорения работы ввода-вывода используется алгоритм отложенной записи (lazy writer) и асинхронная запись на диск.

С точки зрения внутренней организации NTFS - это реляционная база данных, позволяющая протоколировать транзакции, осуществлять поиск по различным атрибутам файлов, производить восстановление данных и т.д. Подробно вопросы внутренней организации и восстанавливаемости будут рассматриваться ниже. И наконец с третьей точки зрения NTFS - это часть общей объектной модели Windows NT, предоставляющая возможность работы с соответствующими файловыми объектами, с контролем доступа и режимами совместной работы. Приложение создает файлы и осуществляет доступ к файлам и каталогам так же, как и ко всем остальным объектам Windows NT: при помощи описателей объектов. Перед получением драйвером NTFS запроса на ввод-вывод, диспетчер объектов и система контроля прав доступа проверяют права вызывающего процесса для доступа к данному файловому объекту. Система контроля сравнивает маркер доступа вызывающего процесса с записями в списке контроля доступа файлового объекта.

2.2.2. Общие принципы организации на диске

Основным понятием в NTFS является *том* (volume). Каждому логическому разделу на диске соответствует свой том. Соответственно путь к любому объекту ФС в NT начинается с имени тома. Каждый том содержит специальные системные файлы, пользовательские файлы, каталоги и специальные области дискового пространства, используемые NTFS. Все системные данные на диске такие как битовые карты, каталоги, системный загрузчик и т.д. NTFS хранит в виде самостоятельных файлов. Базовой и неделимой единицей дискового пространства в NTFS является *кластер*. Как говорилось выше, его размер задается при форматировании и равен от 512 байт до 64 Кбайт (но обязательно степень 2: 1Кб, 2Кб, 4Кб и т.д.). Это число называется *кластерным множителем*. Внутри себя NTFS работает только с кластерами. Все кластеры диска пронумерованы от начала до конца тома и любой физический адрес на диске задается в виде *логического номера кластера* (LCN). Соответственно, физический адрес получается умножением LCN на кластерный множитель.

NTFS отслеживает содержание тома в реляционной базе данных , назывемой MFT (main file table). Как и в любой реляционной базе данных таблица MFT состоит из строк (рядов файловых записей) и столбцов. Каждая строка это запись об одном из элементов ФС: файле или каталоге, а столбцы это атрибуты этого элемента.

NTFS рассматривает файл, не как последовательность байт или записей, а как набор атрибутов, один из которых это данные, содержащиеся в файле. Соответственно, при создании нового файла в MTF добавляется новая запись, а при добавлении новых атрибутов или открытии нового потока данных в соответствующую файловую запись добавляются новые колонки. Как и многие реляционные базы данных NTFS может внутренне организовывать поиск и сортировку по любым атрибутам, но на данный момент реализована индексация и сортировка только по именам. MTF содержит по одной файловой записи для каждого файла тома, включая саму себя, т.к. сама MFT реализована как отдельный файл на диске. Но если файл имеет очень много атрибутов или

сильно фрагментирован, то он может занимать более одной файловой записи, тогда первая запись называется *базовой*.

Файл на томе NTFS идентифицируется 64-битным числом-*файловой ссылкой*. 48 бит отведено под порядковый номер файла в MTF, остальная часть используется для внутреннего контроля целостности в NTFS. Рассмотрим подробно структуру таблицы MTF.

MFT
резервная копия MFT (частичная)
Файл с журналом транзакций
Файл с описателем тома
Таблица определения атрибутов
Корневой каталог
Файл битовой карты
Загрузочный файл
Файл плохих секторов
…
Пользовательские файлы и каталоги

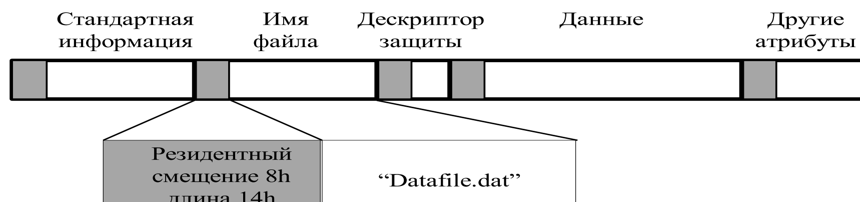
Строка MFT - это файловая запись соответствующая файлу с MFT. Затем идет файловая запись резервной копии MFT, используемой при восстановлении, файловая запись для файла журнала транзакций, файла с данными о томе. Файл с таблицей атрибутов содержит информацию об атрибутах, которые поддерживаются на данном томе. Затем идет запись для корневого каталога ("\"). Файл битовой карты это файл с закодированной информацией о распределении свободных кластеров на диске. Загрузочный файл - это загрузчик системы, а в файл плохих секторов добавляются все обнаруженные системой сбойные кластеры тома. Данные файлы используются только системой и поэтому называются файлами методанных, все остальное являются пользовательскими. Пример участка MFT с пользовательскими данными.

	Стандартная информация	Имя файла	Дескриптор защиты	Данные	Другие атрибуты
Файл 0				Безымянный поток	
Файл 1				Безымянный поток	
Файл 2				Безымянный поток / Имен. поток	
Каталог				Корень индекса / Размещение индекса / битовая маска	
Файл n		MS-DOS имя		Безымянный поток	

(Стандартная информация это набор атрибутов, наследованный от MS-DOS.)

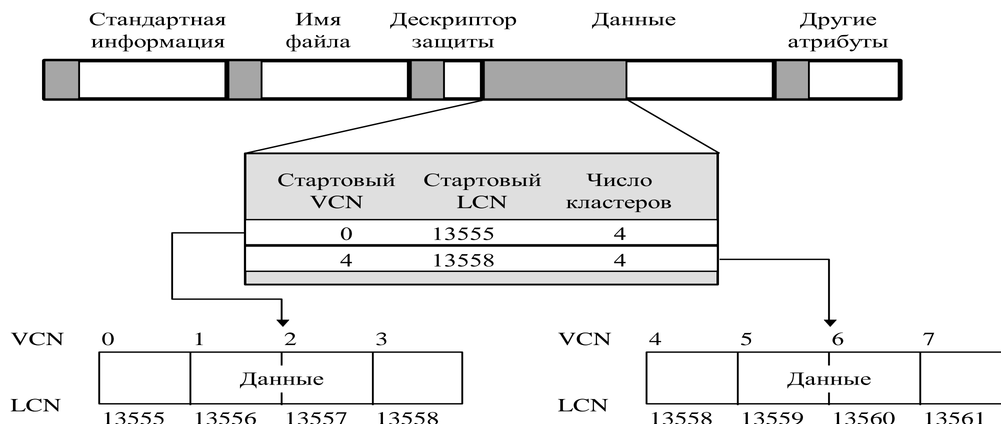
Рассмотрим структуру файловых записей и записей каталогов как элементов таблицы MFT. Размер файловой записи может меняться от 1 Кбайта

до 4 Кбайт и определяется в момент форматирования. Как говорилось выше, каждая файловая запись это набор атрибутов. Все атрибуты упорядочены по типу (точнее по коду типа), некоторые могут встречаться несколько раз (например, если файл имеет несколько атрибутов данных или несколько имен). Атрибуты “Стандартные данные”, “Имя файла”, “Дескриптор защиты” и “Данные” являются обязательными. Значение атрибутов это поток (последовательность) байтов. Для маленьких файлов все его атрибуты и их значения, включая данные, могут храниться в MFT. Атрибуты, значения которых храняться в MFT называются *резидентными*. В начале каждого атрибута содержится его заголовок, который содержит информацию, является ли данный атрибут резидентным или нет. Если атрибут резидентный, то в заголовке так же хранится размер заголовка и размер поля значений данного атрибута. Например:



Если для маленького файла все данные храняться в MFT, то его считывание происходит за одно обращение к диску. Аналогично для небольшого каталога все атрибуты могут быть резидентными.

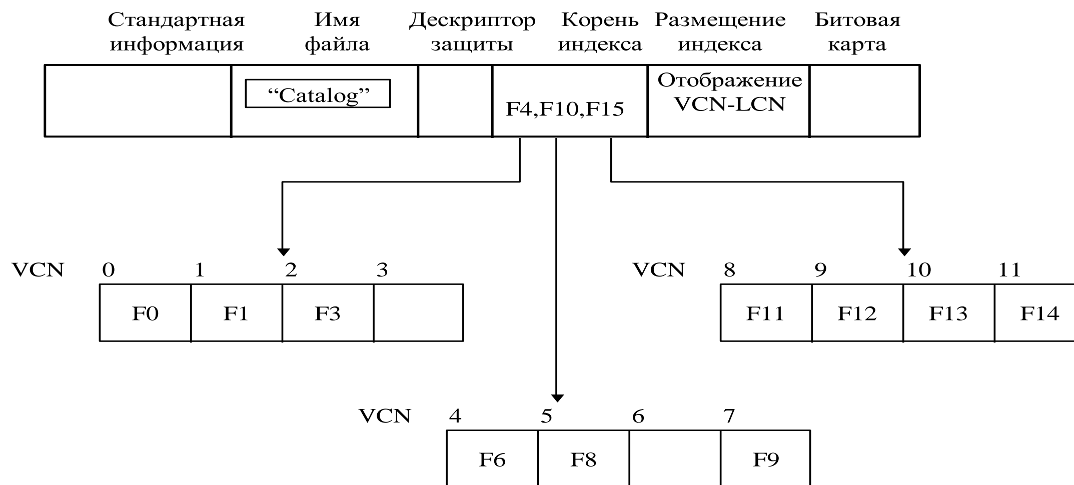
Естественно, что не все данные файла или каталога можно уместить в 1 или 4 Кбайта. Если какой-либо атрибут, например данные не помещается в MFT, то NTFS выделяет для него на диске область равную MAX(кластерный множитель, 2 Кбайта), называемую *отрезком* (extent). Если значение данного атрибута еще увеличилось, то добавляется новый отрезок и т.д. Такие атрибуты называются *неризедентными*. Для связи с отрезками в заголовке неризидентного атрибута задается LCN-VCN (virtual cluster number) отображение. Например:



Если у файловой записи появляется так много атрибутов, что они не помещаются в MFT, то данная запись помечается специальным системным

атрибутом, показывающим, что она является базовой и добавляется ссылка на дополнительную запись в MFT (которая так же добавляется в MTF).

Рассмотрим теперь структуру записи для больших каталогов. Каталог в NTFS это просто индекс имен файлов, т.е. набор имен с файловыми ссылками, организованный в виде сбалансированного дерева для ускорения поиска. Отрезки, выделяемые для хранения элементов индекса называются индексными буферами. Атрибут корня индекса содержит первый уровень индекса и указывает на индексные буферы, содержащие второй уровень и т.д. Для этого с каждым именем файла в индексе связывается помимо файловой ссылки еще и указатель на отрезок, содержащий часть индекса с именами, которые “меньше” в лексиграфическом порядке. Атрибут Размещение индекса содержит VCN-LCN отображения для буферных отрезков, а битовая карта используется для сохранения информации о свободном месте в буферах индекса. Пример для каталога “catalog”:



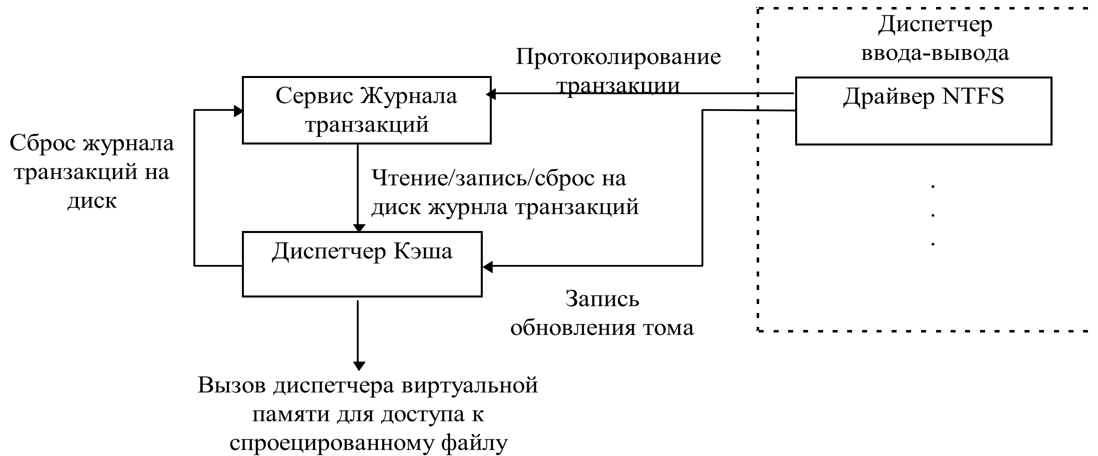
2.2.3. Восстанавливаемость

Одно из наиболее важных свойств NTFS - это ее способность к восстановлению после сбоев. Восстанавливаемость ФС гарантирует, что в случае отказа питания или фатальной системной ошибки ни одна операция ФС не останется незавершенной, а структура дискового тома будет сохранена без использования специальной утилиты исправления диска. Способность к восстановлению придает NTFS дополнительную надежность, причем это достигается за счет лишь незначительного проигрыша в производительности.

Восстанавливаемость ФС в NTFS обеспечивается при помощи техники *протоколирования транзакций* (transaction logging). В процессе протоколирования, прежде чем выполнить над содержимым диска какую-либо операцию, изменяющую важные структуры файловой системы, NTFS записывает информацию о ней в файл журнала транзакций. Тогда в случае сбоя системы можно будет, используя этот журнал, отменить или повторить (если это возможно) все незавершенные транзакции после перезагрузки машины. NTFS определяет транзакцию, как операцию ввода вывода, изменяющую данные ФС или структуру каталогов тома, каждая из которых может состоять из нескольких подопераций.

Средства протоколирования NTFS состоят из двух важных компонент: файла - собственно *журнала транзакций* (log file) и набора системных процедур для доступа к нему - *сервиса журнала транзакций* (log file service, LFS). Отделение

LFS от ФС позволит другим системным компонентам или приложениям создавать собственные журналы для реализации восстанавливаемости на уровне прикладной программы, как это делается в системах обработки транзакций (например в СУБД). NTFS вызывает LFS и предает ему указатель на открытый файловый объект - файл, который будет использоваться в качестве журнала транзакций. LFS, в свою очередь, либо инициализирует новый журнал, либо использует диспетчер кэша Windows NT для доступа к существующему журналу.



При изменении структуры тома NTFS сначала записывает в кэшированный журнал транзакции, модифицирующие структуру тома, затем сбрасывает из кэша на диск журнал транзакций и толко потом записывает на диск из кэша сами изменения тома (т.е. выполняет сами транзакции). Такая последовательность действий гарантирует, что если не удастся выполнить изменения ФС, то соответствующие транзакции можно будет считать из журнала и либо повторить, либо отменить в процессе восстановления ФС (которое инициируется автоматически при первом обращении к тому после перезагрузки машины).

LFS делит файл журнала на две части: *область рестарта* (restart area) и “бесконечную” *область протоколирования* (logging area). В области рестарта NTFS хранит информацию контекста, такую как позиция в области протоколирования, с которой следует начать чтение при восстановлении. На тот случай, если область рестарта будет разрушена или станет по каким-либо причинам недоступна, LFS создает ее копию. В области протоколирования хранятся записи транзакций, обеспечивающие восстановление ФС после сбоя. LFS создает иллюзию бесконечности журнала транзакций путем циклического повторного использования области протоколирования, однако при этом гарантируется, что нужная информация не будет затерта. Для идентификации записей используются *номера логической последовательности* (logical sequence number, LSN), которые увеличиваются, несмотря на циклическое использование журнала. Однако максимально возможный LSN настолько велик, что логический размер журнала можно считать практически бесконечным.

Прежде чем модифицировать том, NTFS обращается к LFS для регистрации в журнале соответствующих транзакций, что отвечает технике *опережающего протоколирования* (write-ahead logging). NTFS использует два типа записей - *записи модификации* (update record) и *записи контрольной точки* (check-point record). Каждая запись модификации состоит из двух частей:

- *Информация для повтора* (redo information) - как вновь применить к тому подоперацию подтвержденной транзакции в случае, если сбой произошел до того, как транзакция была переписана из кэша на диск.
- *Информация для отмены* (undo information) - как устранить изменения, вызванные подоперацией транзакции, которая на момент сбоя была запротоколирована лишь частично.

После того, как все подоперации транзакции были запротоколированы с помощью записей модификации, NTFS выполняет транзакцию в кэше. По окончании обновления кэша в журнал помешается еще одна запись, означающая, что транзакция завершена - *подтверждение транзакции* (commit a transaction). NTFS гарантирует, что модификации тома подтвержденной транзакции будет выполнены даже в случае сбоя ОС.

Для восстановления диска NTFS использует две таблицы, постоянно поддерживаемые в памяти - *таблицу транзакций* (transaction table), содержащую LSN всех неподтвержденных на данный момент транзакций, и *таблицу измененных страниц* (dirty page table), содержащую информацию о том, какие страницы кэша содержат не записанные на диск изменения структуры ФС. Каждые 5 секунд NTFS обращается к LFS для сохранения текущей копии этих таблиц в файле журнала транзакций. Затем в область протоколирования помещается запись контрольной точки, а в область рестарта записывается LSN этой записи. Таким образом, на момент сбоя в журнале сохранены самые последние копии таблиц транзакций и измененных страниц, запись соответствующей им контрольной точки (причем ее LSN записан в области рестарта) и несколько записей модификации, внесенных в журнал уже после выставления контрольной точки.

При восстановлении тома NTFS выполняет три прохода по журналу транзакций, загружая его в память на первом проходе, чтобы минимизировать объема дискового ввода-вывода. Во время *прохода анализа* (analysis pass) журнал просматривается в прямом направлении, начиная с последней операции контрольной точки с тем, чтобы обновить таблицы транзакций и измененных страниц, используя сохраненные для этой контрольной точки копии. Кроме того, производятся все модификации таблиц, соответствующие записям журнала, помещенным туда уже после записи контрольной точки. При этом, если среди этих записей встречается запись “подтверждение транзакции”, то эта транзакция удаляется из таблицы транзакций. Таким образом, проход анализа восстанавливает содержимое таблиц на момент сбоя.

На *проходе повтора* (redo pass) NTFS сканирует журнал в прямом направлении, начиная от LSN последней не сохраненной на диск записи модификации (найденной на проходе анализа). Для всех найденных записей “обновление старницы”, содержащих модификацию тома, NTFS повторяет операции в кэше, для чего используется информация для повтора соответствующих записей журнала. Когда NTFS достигает конца журнала, она уже обновила кэш всеми необходимыми модификациями тома (выполнила все подтвержденные транзакции).

Завершив проход повтора, NTFS начинает *проходе отмены* (undo pass), просматривая журнал в обратном направлении и отменяя все подоперации неподтвержденных транзакций, LSN которых берется из уже восстановленной таблицы транзакций, для чего используется информация для отмены соответствующих записей журнала.
Когда проход отмены завершен, целостность тома восстановлена. В этот момент NTFS сбрасывает на диск изменения кэша с тем, чтобы гарантировать правильность содержания тома. После этого NTFS записывает пустую область рестарта, указывающую, что том находится в нормальном состоянии и что в случае нового сбоя восстановления не требуется. Заметим, что NTFS гарантирует восстановление тома к некоторому существовавшему ранее целостному состоянию, но не обязательно к тому, которое непосредственно предшествовало сбою, т.к. сброс журнала транзакций из кэша на диск выполняется не всякий раз при получении записи “транзакция завершена”, а сразу для пакета из нескольких завершенных транзакций.