Содержание

1 Лекция 1

1.1 Файловая подсистема Linux

Файловая подсистема предназначена для обеспечения возможности хранения и доступа к файлам в системе. Это задача которая стоит перед любой файловой системы любой ОС. Unix определил подходы к построению тех или иных модулей системы, к структуризации системы в целом.

Определение файла из Оксфордского словаря: файл - информация хранимая во вторичной памяти или во вспомог уст с целью ее созранения после заверш отдел задания или преодал огр основного зап устройства (т.н. рабочие файлы). В файле может содержаться любая информация.

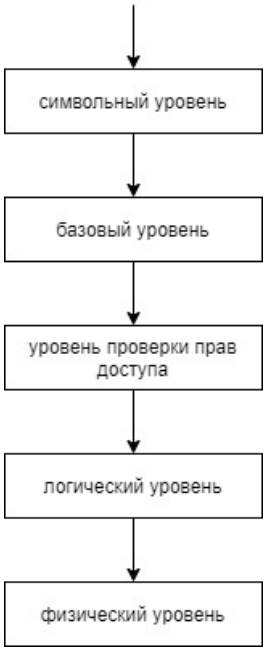
Современное определение файла: файл - это любая поименованная совокупность данных, которая хранится во вторичной памяти.*

Определение файловой системы: файловая система - это порядок, определяющий способ организации хранения, именования и доступа к данным на вторичных носителях информации. (Определение состоит из указания задач файловой системы).

Это общие представления о файловой системе.

Рассмотрим обобщенную модель файловой системы. (Из иерархической модели Медника-Донована - на самом высоком уровне файловая система). Аналогично любая файловая система имеет иерархическую структуру. Это связано с разными уровнями этой файловой системы так как различные задачи, которые решает файловая система, выполняются на разных уровнях операционной системы.

Именование файлов (символьный уровень) - это самый высокий уровень файловой системы. Он позволяет пользователю в удобной форме задавать имена файлов и искать их в каталогах. Но файл хранится на физическом уровне (является внешним), значит на нижнем уровне доступ к нему осуществляется с подсистемой ввода-вывода, так как, чтобы считать файл необходимо обратиться к внешнему устройству.



В системе у файла существует полное имя и короткое имя. Система всегда оперирует полным именем. В различных системах имеются отличия в представлении этого имени (в Windows оно начинается с имени логического диска, в Linux - с корневого каталога).

В Unix имя файла не является его идентификатором. В системе файл идентифицируется номером inode'а. Inode - (фактически) дескриптор файла. Первые два уровня определяют именование файла в системе (см рис 1).

Затем модуль проверки прав доступа. (Важнейшая задача ос - контроль прав доступа read/write/execu

Логический уровень. В данном случае файл похож на программу. Любая программа считает, что она начинается с нулевого адреса. В программе находится смещение. Логическая организация файла начинается с нуля.

Физический уровень. Файл хранится на внешнем устройстве - это уже подсистема ввода-вывода - это уже прерывания. На этом уровне осуществляется учет особенностей организации внешнего устройства.

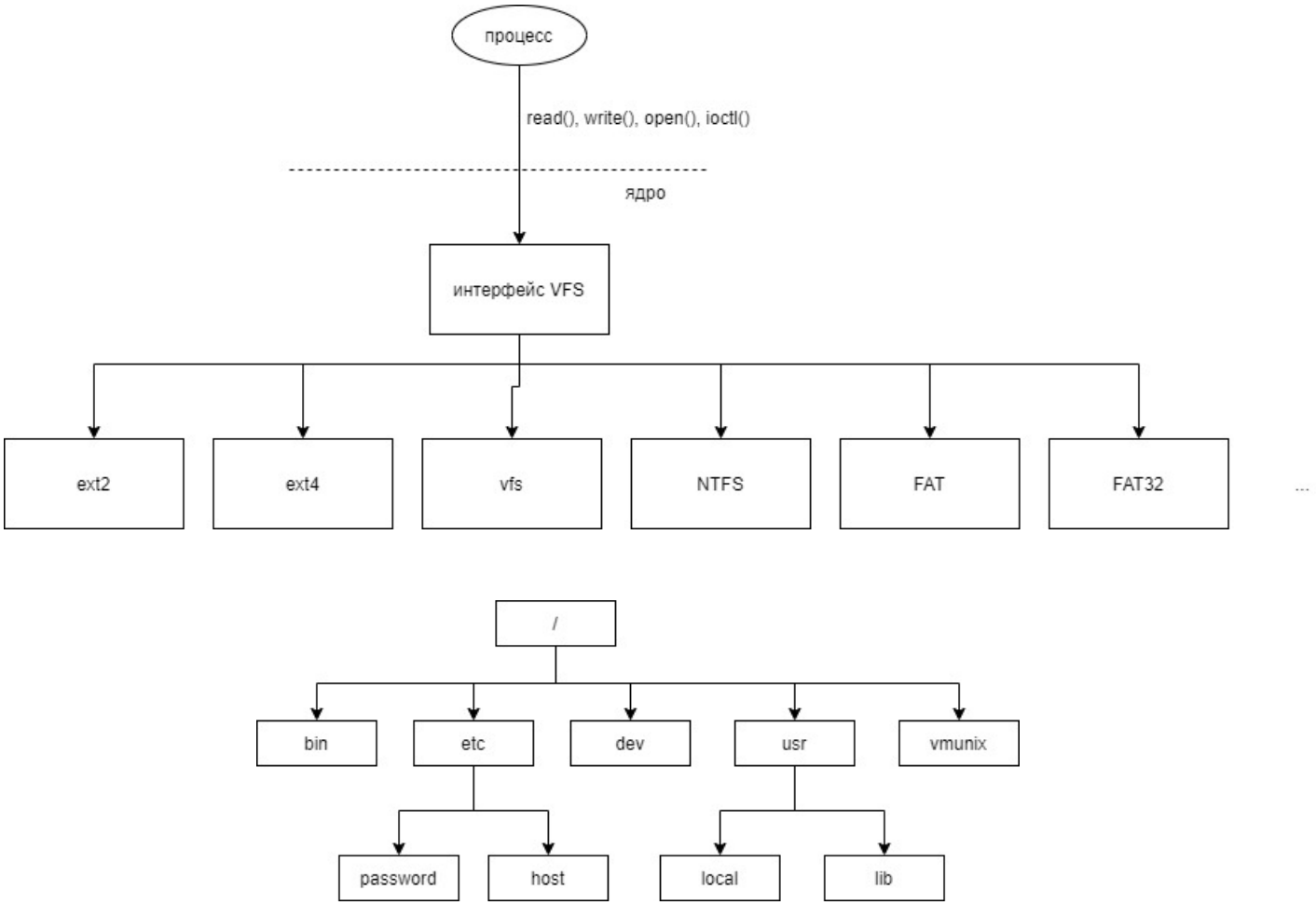
Дисковые устройства на сегодняшний день являются единственными блочными устройствами (и флэш-память). Все остальные устройства - символьные.

Это самое общее представление уровней файловой системы.

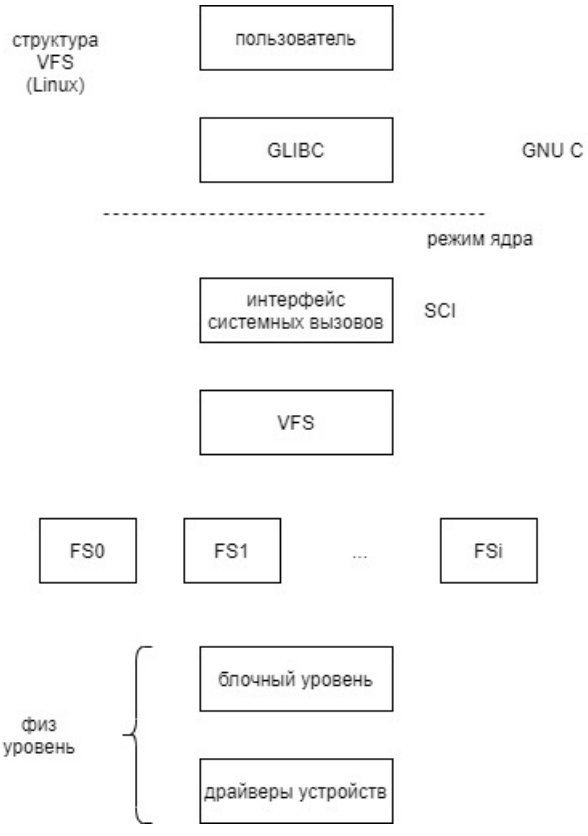
В UNIX Linux имеется существ отличия в системе для работы с файлами.

В UNIX для работы с файлами организовано через интерфейс, который называется VFS/vnode (Virtual file system/virtual node).

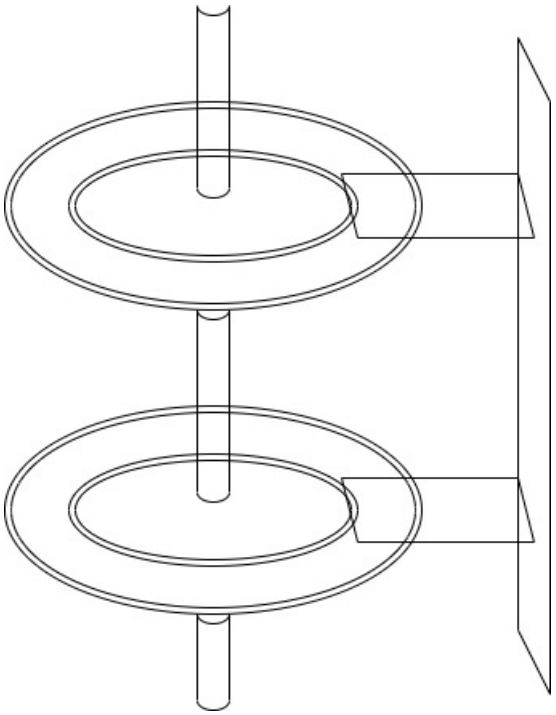
В Linux VFS не определена структура vnode. Это сделано для того, чтобы обеспечить широкую поддержку различных файловых систем без перекомпиляции ядра.



Характерное для UNIX/Linux дерево каталогов.



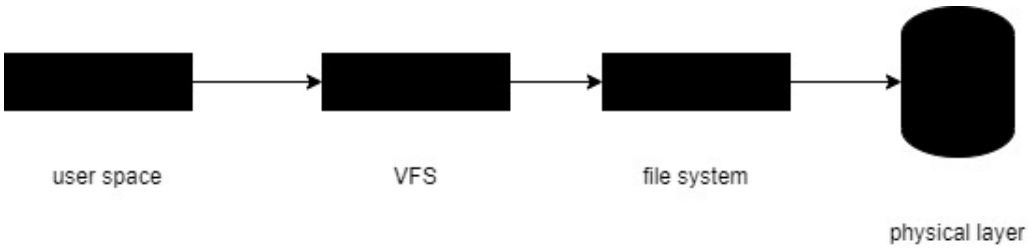
Для Linux важными являются аббревиатуры GNU C и SCI (System call interface / Standard system call interface).



Магнитные диски.

Современные файловыми системами поддерживаются так называемые очень большие файлы. Проводя аналогию с управлением памятью (таблицами страниц), обеспечивается хранение файлов в разброс, они не занимают непрерывные адреса в памяти. Обеспечивается возможность хранения информации из одного и того же файла и это обеспечивается хранением соответствующей информации, а именно адресов блоков.

Такая возможность в ОС юникс линукс реализуется за счет того, что в состав системы входит "слой абстракции"над собственным низкоуровневым интерфейсом файловой системы. Для этого VFS предоставляет общую файловую модель, которая способна отображать общие возможности и "поведение"любой возможной файловой системы. Такой уровень абстракции работает на основе базовых концептуальных интерфейсов и структур данных, которые поддерживаются конкретными файловыми системами. Фактический код любой файловой системы скрывает детали реализации непосредственной работы с данными, организованными в файлы. А именно, предоставл в распоряж польз как правило набор API, таких как открыть файл, прочитать файл, записать в файл, удалить файл, переименовать файл и т.д. Любая файловая система поддерживает такие понятия как файл, каталог и опять же, действия опред над файлами (уже перечисленные). Это можно представить себе следующей абстракцией.

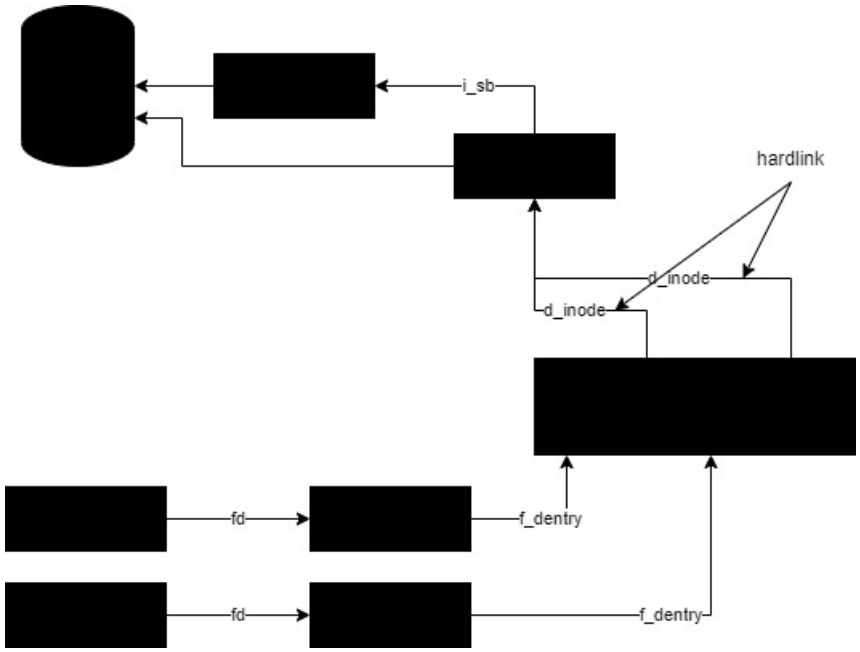


VFS предоставляет общую файловую модель, которую наследуют низлежащие файловые системы, реализуя действия для различных Posix API.
Системный вызов write() сначала обрабатывается общим системным вызовом sys_write(), который определяет фактический способ записи, характерный той фс, на которой находится файл. Затем общий системный вызов sys_write() вызывает метод конкретной фс для того, чтобы выполнить запись данных на физический носитель. То есть, VFS (как любая фс) скрывает особенности работы с конкретным физическим устройством.

1.2 Организация VFS в Linux

Внутренняя организация VFS построена на 4-х структурах:
1. Суперблок (superblock)
2. Индексный узел (inode)
3. Запись каталога (dentry)
4. Файл (file)

Есть диаграмма связи этих структур.



Система различает файл находящийся на диске (его описывает inode) и открытый файл (его описывает struct file) это и показано на рисунке. Показана связь всех 4 структур.

Показано, что 2 процесса открыли файл (один и тот же с разными именами). У него (файла) 1 inode. Очевидно, чтобы обесп доступ к этому файлу в системе имеется struct dentry (сокращение от directory entry) – обеспечивает работу с каталогами.
Структура superblock в VFS описывает конкретную подмонтированную файловую систему. Т.е. файловую систему, которая находится во внешней памяти, которая располагается на внешнем носителе (подмонтированная).
f_dentry, dinode, i_sb - это не идентификаторы, это имена полей соответствующих структур. В struct file есть поле dentry, которая ссылается на структуру, описывающую соответствующий каталог и т.д.
Любой файл хранится в (=принадлежит) конкретной (рабочей, т.е. определенной на диске) файловой системе.

1.3 Суперблок

Суперблок - это контейнер для высокоуровневых метаданных о файловой системе.
Суперблок - структура, которая должна находится на диске, тк эта стр опис фс. Для надежности он хранится в нескольких местах диска. В этой структуре хранятся управляющ параметры фс, такие как суммарное число блоков, свободное число блоков, корневой inode и т.д.
В системе сущ суперблок на диске и суперблок в оперативной памяти. Суперблок на диске хранит информацию необходимую системе для доступа к физическому файлу, т.е. хранит информацию о структуре файловой системе. Суперблок в памяти предоставляет информацию, необходимую для управления смонтированной файловой системой. При этом в системе имеется связный список - struct list_head superblocks (хранит информацию обо всех подмонтированных файловых системах). Эта структура определена в <linux/fs.h>.

1.4 Монтирование

В дерево файлов и катагов входят отдельн ветви, которые формируются разными фс. Для Юних определено подключение различных файловых систем в дерево каталогов и это действие получило название монтирование.
Монтирование - система действий, в результате которых файловая система устройства становится доступной т.е. можно получить доступ к информации, которая хранится на устройстве.
Базовая форма команды mount принимает на вход 2 параметра: имя устройства (или какого-то другого ресурса, которая содержит монтируемую фс) и точку монтирования. Могут присутствовать ключи - тип файловой системы, опции файловой системы.
mount ключи -t тип_фс -o опции_фс устройство каталог(=точка_монтирования)

2 Лекция 2

2.1 Монтирование и файловые системы

Путь файла начинается с корневого каталога, который обозначается /. Монтирование - система действий, в результате которой файловая система становится доступной. Для монтирования требуются права и привелегии пользователя. Для монтирования используется команда mount, имеющая следующий синтаксис:
mount ключи -t тип_файловой_системы -o опции_файловой_системы устройство каталог_назначен

Для размонтирования приеняется команда unmount:
unmount ключи -t тип_файловой_системы -o опции_файловой_системы

Кроме основной команды существуют дополнительные команды, в которых например может указываться имя файловой системы, например mount_nfs (Network File System). Если подмонтирована файловая система windows, то mount_ntfs.
Наиболее часто в команде mount используется два параметра - имя устройства, или другого ресурса, который содержит монтируемую файловую систем и точку монтирования.
Точка монтирования - каталог к которому подмонтируется файловая система. Точка монтирования должна существовать, иначе возникнет ошибка.
Когда файловая система смонтирована в существующую директорию, все файлы и поддиректории этой смонтированной файловой системы становятся файлами и поддерживают точку монтирования.
Если директория точки монтирования содержала в себе какие-либо файлы и поддиректории, то они не теряются, а становятся невидимыми.
Иногда может оказаться нужным явно указывать при монтировании тип файловой системе. Для этого в команде mount используется опция -t. Это нужно для того, чтобы отследить попытку монтирования файловой системы, использующую новый тип. Unix/Linux могут поддерживать большое количество файловых систем. Существует структура, описывающая тип файловой системы.
Рассмотрим пример:
mount /dev/sda1 /mnt - опции -t -o отсутствуют, данная команда пробует монтировать раздел sda1 с файловой системой ext3 в каталог /mnt в режиме только чтения. Если в системе нет библиотек для работы с той или иной файловой системой, или система в указанном разделе не является ext3, будет выдано сообщение о невозможности монтирования.
Если требуется включить режим записи, то необходимо добавить # mount -o rw /dev/sda1 /mnt.

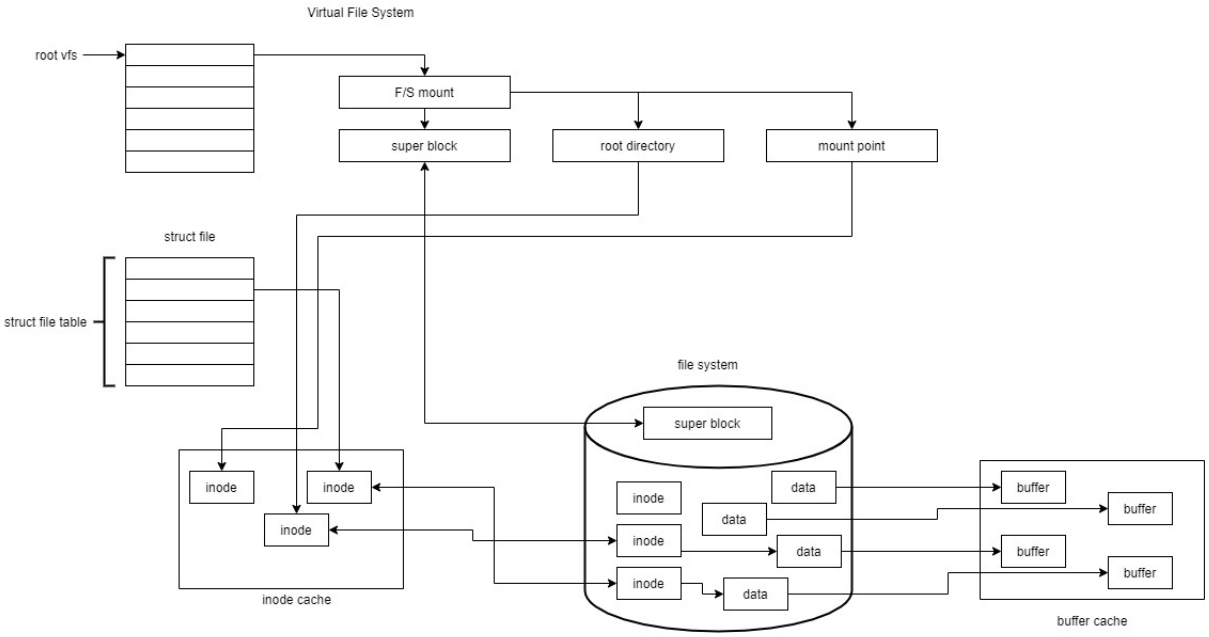


Рисунок 2.1 — Схема работы файловой системы

Структура superblock предназначается для подмонтированных файловых систем. Данная структура содержит всю необходимую информацию для обращения к файлом конкретной файловой системы. struct inode - струткура физического файла. struct ientry - структура, описывающая элемент каталога и предназначена для доступа к файлам. struct ifile - структура, описывающая открытый файл, при этом открытый файл - это файл, который открыт каким-то процессом.
Пользователя для системы не существует. Для системы существуют только процессы, которыми она управляет.
inode - два варианта, не являющиеся копиями. В ядре существуют структуры, важные для действий в ядре. Мы видим, что inode существует дисковый и ядре. Нужна точка монтирования, нужен корневой каталог.
Есть кеш inode и буферный кеш (кеш данных), в системе всё буферизируется. Кеши построены по принципу LRU (last recently used). Поскольку они не могут быть любого нужного размера (ограничены возможностями физического хранения в ядре). По своему назначению - хранят данные, к которым были последние обращения. Также здесь присутствует системная таблица открытых файлов. В этой системной таблице находятся дескрипторы всех открытых в системе файлов. Причём если файл был открыт несколько раз, в этой таблице будет существовать соответствующее количество дескрипторов открытого файла. Так ак открытые файлы - это структура, предназначенная для обслуживания процессов (процессы открывают файлы). При этом разные процессы могут открывать один и тот же файл. У этого файла может быть один и тот же inode (существуют hard линки) - система не различает имена файлов (первое, второе и тп.) Все имена системы - хардлинки.

2.2 Суперблок

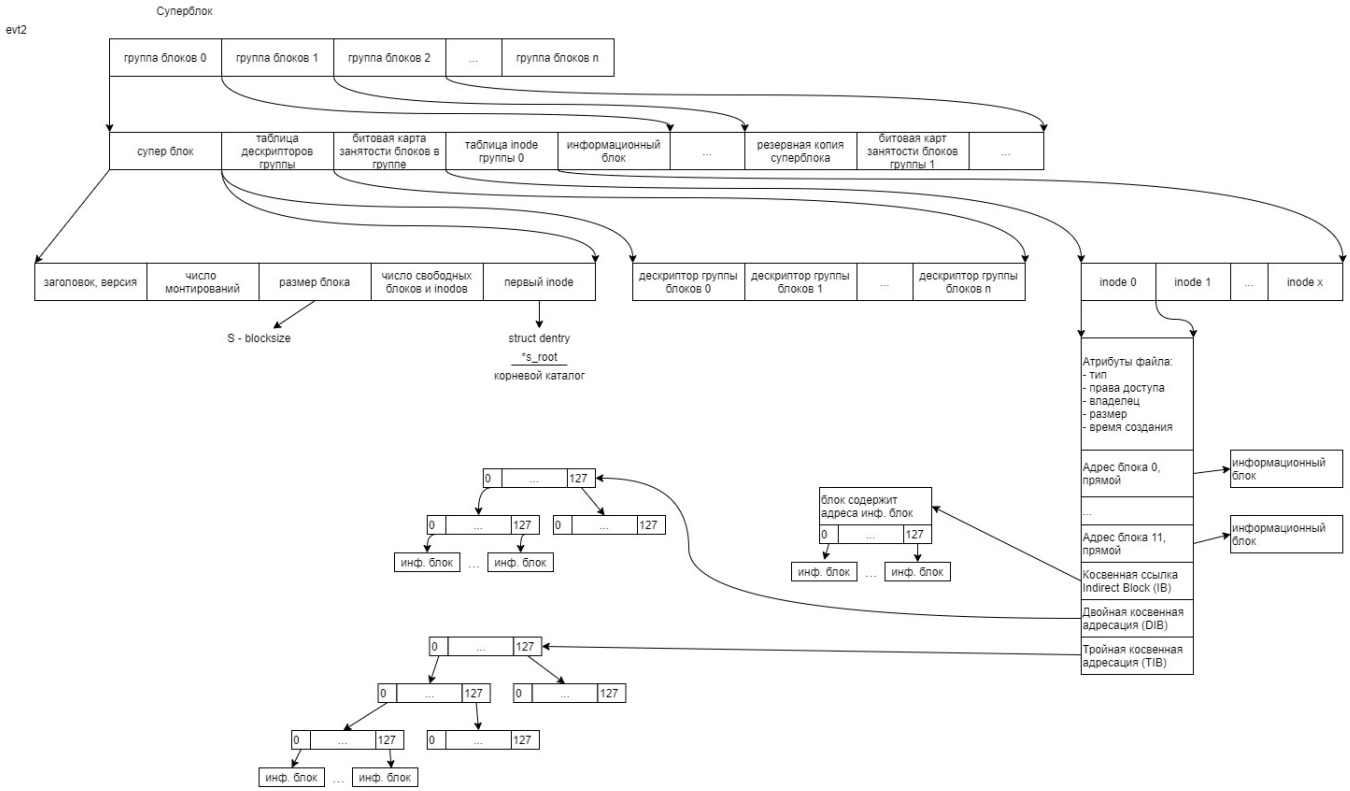


Рисунок 2.2 — Схема работы суперблока

В данном примере мы рассмотрим раздел жёсткого диска (раздел вторичной памяти) с файловой системой ext2 (родная файлова система linux).

Блок - минимальная адресуемая единица физического носителя (вторичной памяти). В разных системах размер может отличаться. Каждый блок имеет уникальный адрес.

Любая файловая система предназначена для обеспечения долговременного хранения и доступа к файлам. Без доступа все это не нужно. Кроме того, что файловая система предназначена для хранения информации, однако в первую очередь она должна обеспечивать доступ к этой информации. Доступ - многоуровневый (вторичная память - внешнее устройства) на каком-то последнем этапе происходит обращение к устройству.

inode описывают физический файл, и они же так называемые дисковые inodeы содержат информацию об адресных блоках вторичной памяти, в которой находится файл, который описывается конкретным inodom.

Очевидно, что суперблок должен содержиться список inodов, но он должен содержать первый inode - inode корневого каталога.

Файловая система ext2 существует достаточное большое количество времени по причине того, что она хорошо написана. Данная система обеспечивает хранение и доступ к очень большим файлам. По аналогии с физическим адресным пространством оперативной памяти, если первые операционные системы требовали. чтобы процессу было выделено непрерываное адресное пространство, и это оказалось невозможным при быстром росте прикладного ПО. Такое требование

в современных системах физически обеспечить невозможно. Не существует достаточно большого объекма физической памяти для обеспечения одновременной работы большого объема фзической памяти для обеспечения одновременной работы большого количества приложений с высокими требования к физической памяти, что характерно и для файлов. Размеры файлов также постоянно растут. Это было понято разработчиками Linux/Unix. Ими было предложено решение, когда файлу выделялось не непрерывное адресное пространство, а была обеспечена возможность выеления свободных участков адресного пространства. В результате было обеспечено хранение и доступ к файлам очень большого размера.

Практически все источники приводят эту информацию о файловой системе ext2 как наиболее яркий пример работы с файлами очень большого размера.

Видно, что суперблок хранит информации о inodax, обеспечивая доступ к дисковому inody, а дисковый inode хранит информацию об адресных блоках вторичной памяти, в которых располагается данные данного файла.

В этой файловой системы существует несколько типов адресации - прямая, косвенная, двойная косвенная и тройная косвенная.

Для адресов 12 блоов (0 - 11) используются для прямой адресации. Соответственно это непосредственно адрес информационного блока.

Следующие блоки содержат косвенные ссылки (indirect block). Этот блок содержит адреса инфомационных блоков.

```
1  struct super_block
2  {
3      struct list_head head s_list;
4      dev_t s_dev;
5      unsigned long s_blocksize;
6      struct file_system_type * s_type;
7      unsigned long s_flags;
8      unsigned long s_magik;
9      struct dentry *s_root;
10     struct rw_semaphore s_unmount;
11     ...
12     struct list_head s_mount;
13     struct block_device *s_bdev;
14
15     const struct dentry_operations *s_d_op;
16     ...
17     /* s_node_list_lock protects s_inode*/
18     spin_lock_t s_inode_list_lock;
19     struct list_head s_inodes;
20     ...
21     struct list_head s_inodes_wb;
22 };
```

Суперблоки объединены в списокк. В системе будет существовать столько суперблоков, сколько смонтировано файловых систем. При этом может быть смонтировано несколько файловых систем одного и того же типа, поэтому первое поле представляет из себя список.
Очевидно, что смотированая файловая система должна находиться на каком-то девайсе или на части ос, на части адресного пространство оперативной памяти или на внешнем устройства (это должен быть физический носитель) - второе поле.
Важное значение имеет размер блока (он является минимальной адресной единицей на вторичной памяти) - третье поле.
Тип файловой системы - важнейшее понятие в Linux, так как может поддерживать большое количество файловых систем. Структура file_system_type предназначена для регистрации типа конкретной файловой системы:
Есть флаги для работы с файловой системой
Есть магическое число для обеспечения надёжной работы
Указатель на root
Семафоры для чтения-записи

На суперблоке определён набор операций, которые с ним можно выполнять - struct super_operations.
В любой из перечисленных структур есть ссылка на операции, которые могут выполняться на этом суперблоке, фактически таблицу. Дентри являеся элементом пути к файлу. Каждый элемент пути имеет элемент inode и хранится на диске. Каждая структура содержит средства взаимоисключения. Список inodов должен защищаться, для этого используется спинлок.
Кроме этого есть список inodов. В этом списке находятся все inodы - дескрипторы физических файлов, созданных в конкретной файловой системе.
s_inodes_wb - Write Back inodes, список грязных inodов. Грязным файлом называется изменённое значение.

```
1  struc super_operations
2  {
3      struct inode * ( *alloc_inode)(struct super_block *sb);
4      void ( *destroy_inode)(struct inode *);
5      void ( *free_inode)(struct inode *);
6
7      void ( *dirty_inode)(...);
8      int ( *write_inode)(...);
9      int ( *drop_inode)(...);
10     int ( *evict_inode)(...);
11
12     void ( *put_super)(struct super_block, ...);
13     int ( *remount_fs)(struct super_block, ...);
14     void ( *unmount_begin)(struct super_block, ...);
15
```

```
16 };
```

Очевидно, что поскольку любая файловая система предназначена для хранения физических данных. Причём не просто данных, а поименованных. Мы видим, что это работа с inodom.

3 Лекция 3

3.1 Структура operations

Функции в структуре operations описаны в мануале unix.

Очевидно, что суперблок это основная структра, с которой начинается доступ к конкретной файловой системе. Когда файловая система монтируется, то на диск загружается её суперблок. Важнейшей информацией являются указатели на списки i_nodeов - структур, описывающих физические файлы на диске и содержат информацию для доступа к информации, хранящейся в конкретном файле.

dirty - название для изменённых элементов файловой системы. Данная информация является важной для системы, поскольку таким образом система помечает для себя, что данны структуры должны быть обновлены.

drop_inode - вызывается подсистемой ВФС, когда исчезает последняя ссылка на inode.

write_inode - функция для освобождения inoda.

put_super - вызывается при размонтировании файловой системы.

remount_fs - вызывается при монтировании с другими параметрами монтирования.

Новый суперблок создаётся командой alloc_super.

```
1  static struct super_block *alloc_super(struct file_system_type * type, int flags,
       struct user_namespace *user_ns)
2  {
3      // Вызов, выделяющий память
4      struct super_block *s = kzalloc(sizeof(struct super_block, GFP_USER));
5
6      // Определение структуры struct_superoperations
7      static const struct super_operations default_op;
8  }
9
10 static void destroy_super(struct super_block *s){...}
```

Если файловой системе необходимо выполнить запись в суперблок, то будет вызвана функция write_super следующим образом:

```
1  sb->s_op->write_super(sb);
```

В ВФС определена структура file_system_type, описывающая конкретный тип файловой системы. Рассмотрим её подробнее. Файловая структура одна, смонтированных файловых систем может быть много.

```
1  struct file_system_type
2  {
3      const char *name;
4      int fs_flags;
5      #define FS_REQUIRES_DEV 1 // определяется требование блочного устройства
```

```
6       #define FS_USERNS_MOUNT 8 // определяет, что файловая система монтируется на
            root user
7
8       struct dentry * ( *mount)(struct file_system_type*, int, const char*, void *);
            // для монтирования и заполнения суперблока соответствующими данными.
9       void ( *kill_sb(struct super_block*);
10
11      struct file_system_type *next;
12      struct file_system_type *next;
13      struct hlist_head fs_super; // список объектов типа super block
14
15      struct lock_class_key s_lock_key;
16      struct lock_class_key s_unmount_key;
17      // и ещё 5 подобных полей
18  }
```

Многие поля в file_system_type отвечают за предоставление монопольного доступа к элементам файловой системы.

Виртуальная файловая система не связана ни с каким блочным устройством. Поэтому при написании лабораторной по виртуальной файловой системе необходимо объявить тип файловой системы.

Когда файловая система монтируется, для конкретной файловой системы выделяется экземпляр структуры struct_vfs_mount. Эта структура представялет конкретный экземпляр файловой системы.

```
1  struct vfsmount
2  {
3      struct dentry *mount_root; // точка монтирования
4      struct super_block *mnt_sbl // указатель на суперблок
5      int mnt_flags; // флаги монтирования
6  };
```

Пример заполнения стурктуры file_system_type:

```
1  struct file_system_type my_fs_t =
2  {
3      .owner = THIS_MODULE,
4      .name = "my_fs",
5      .kill_sb = my_kill_superblock,
6      .fs_flags = FS_REQUIRES_DEV
7  }
```

Одна и та же файловая система может быть подмонтирована несколько раз. В системе определено несколько разновидностей фунцкии mount.

```
1  extern struct denty *mount_ns(struct file_system_type *fs_type, int flags, void
       data, int ( *fill_super)(struct super_block *, void *, int));
```

```
2 extern struct dentry* mount_bdev(struct file_system_type* fs_type, int flags, const
    char* dev_name, void* data, int ( *file_super)(...);
3 extern struct dentry *mount_nodev(struct file_system_type* fs_type, int flags, void*
    data, int ( *file_super)(...));
```

fill_super выполняет основную работу по заполнению полей в структуре super_block.

Для того, чтобы операционная система и виртуальная файловая система/подсистема операционной системы зарегистрировала созданную структуру file_system_type в системе имеются соответствующие функции.

```
1 res = register_filesystem(&my_fs_t);
```

Это делается в функции init модуля. В результате системе станет известно имя вашей функции mount, kill_super.

```
1 res = unregister_filesystem(&my_fs_t); // вызывается в функции exit модуля
```

3.2 Индексный дескриптор inode

В UNIX имя файла не является его идентификатором. Идентификатором файла является его inode. Соответственно у каждого файла есть один inode. В системе существует два типа inodов. Дисковый inode и inode ядра. Очевидно, что описывая один и тот же файл эти структуры противоречить друг другу не могут, однако дисковый inode содержит информацию об адресах блоков данных конкретного файла. Но для сокращения обращений к данным файла и подобным данным всё кэшируется. Именно отсюда следует, что в системе существует inode ядра, к которому происходит обращение при работе с файлом. Это делается для ускорения доступа к данным.

Копии индекса, которые находятся в памяти содержат поля, которых нет в дисковом inode, а именно:

1) поля, ответственные за блокировки

2) логический номер устройства

3) номер индекса (в дисковом индексе это поле не нужно, так как на диске индексы хранятся в линейном массиве и ядро идентифицирует индекс по го смещению)

4) ссылки на другие индексы для организации в ядре хеш-очереди. Кроме этого ядро ведёт список свободных индексов.

5) счётчик ссылок на файлю Т.е. сколько раз файл был открыт.

```
1 struct inode
2 {
3     umode_t i_mode;
4     kuid_t i_uid;
5     kgid_t i_gid;
6     unsigned int i_flags;
```

```
    ...
    const struct inode_operations *i_op;
    struct super_block *i_sb;
    unsigned long i_ino;

    union {
        const unsigned int i_nlink;
        unsigned int _i_nling;
    }

    dev_t i_rdev;
    uoff_t i_size;
    ...
    unsigned short i_bytes;
    unsigned int i_blksize;
    blkcnt_t i_blocks;
    ubu i_version;
    atomic_t i_count;
};
```

Операции в системе, определённые для inodов:

```
struct inode_operations
{
    struct dentry * ( *lookup)(struct inode *, struct dentry *, unsigned int);
    const char * ( * get_link)(struct dentry *, struct inode*, struct delayed_call*
        );
    int ( *permission)(struct inode*, int);
    ...
    int ( *link)(struct dentry*, struct inode*, struct dentry*);
    int ( *symblink)(struct inode*, struct *dentry *, const char *);
}
```

4 Лекция 4

struct dentry описывает элемент пути начиная с корневого каталога.

/home/dracula/src/foo.c - для того, чтобы система обратилась к файлу foo.c необходимо, чтобы она спустилась по всем элементам пути. Каждый элемент каталога это файл, иначе невожножно хранить информацию о дереве каталогов, которое позволяет получить доступ к файлам. То есть каждый элемент такого каталога будет иметь inode, и это реальный inode. Эти элементы с точки зрения структуры struct dentry создаются налету.
Очень показательным моментом является момент в библиотеке <linux/dcach.h>. При этом в этом кэше кешируются inodы, которые представляют элементы пути. А любой элемент пути - файл. Это делается для того, чтобы сократить времмя доступа к файлам. На прицнипе LRU строится кеширование.
Пользователь не может ждать неопределённо долго, при постоянном ожидании накапливается волнение. Время ответа системы не должно превышать трёх секунд.
Кэш дентри состоит из трёх частей:
Список объектов dentry которые связаны с определённым inode поскольку в struct inode есть поле i_dentry. Один и тот же inode может иметь несколько ссылкок, соответствующих директориям, то это поле должно представлять из себя связный список.
Двухсвзяный список неиспользуемых и противречивых объектов dentry по алгоритму LRU. Очевидно, что если это алгоритм LRU, то добавление элемента в двухсвязный список нового объекта dentry должно выполняться по значению времени. Новый объект dentry записывается в хвост, удаление выполняется из головы. Очевидно, что при работе с файлами нет диких обращений к файлам, поэтому алгорит LRU может быть использован.
Хэш-таблица или хэш-фукнция, которые позволяют преобразовывать данный путь в объект dentry. Данный объект называется dentry_hashtable. Каждый объект таблицы является указателем на объект dentry, который соответствует какому-то ключу. Значение ключа определяется функцией d_hash(). Что позволяет для каждой файловой системы реализовать свою хэш-функцию. Поиск в хэш-таблице осуществляется функцией d_lookup().

Приведённый пример показывает, что сначала идёт поиск в dentry кэше, если поиск приводит к тому, что какого-то элемента каталога нет в кеше, то тогда идёт обращение таким образом, который мы рассмотрели. В итоге найденный объект будет помещён в кэш. Struct dentry имеет указатель на dentry operations.

```
1 struct dentry.operations
2 {
3     int ( * d_hash)(const struct dentry * , struct qstr * );
4     int ( * d_compare)(const struct dentry  * , const struct dentry * , unsigned
          int, const char * , const struct qstr * );
5     int ( * d_delete)(const struct dentry * );
```

```
6       struct vfsmount * ( * d_automount)(struct path * );
7   }
```