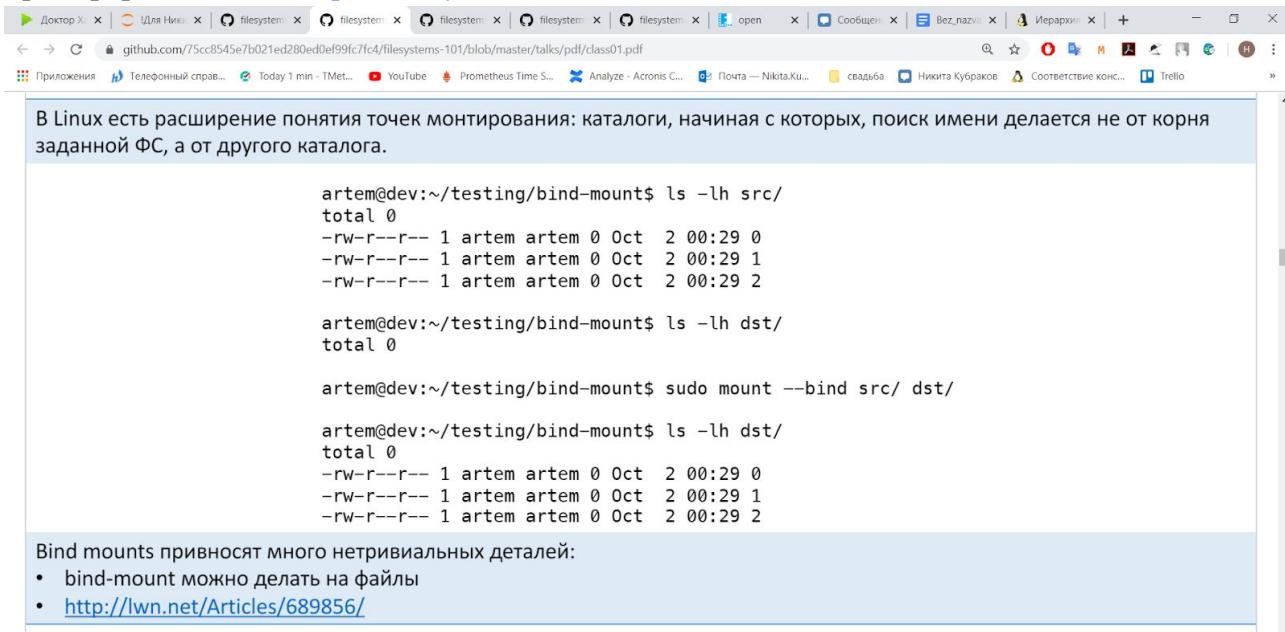


5.

Про иерархию и FHS <http://linux.yaroslavl.ru/docs/conf/fs/fhs-full.html>



B Linux есть расширение понятия точек монтирования: каталоги, начиная с которых, поиск имени делается не от корня заданной ФС, а от другого каталога.

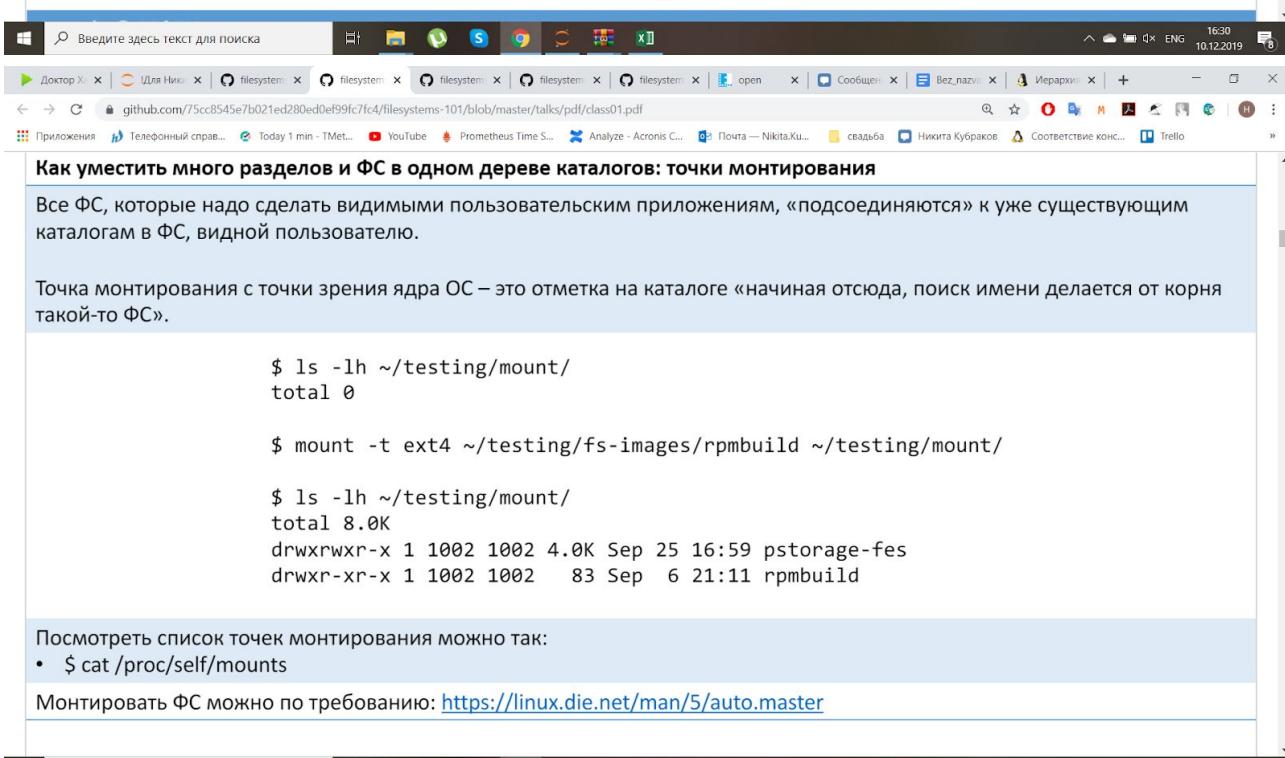
```
artem@dev:~/testing/bind-mount$ ls -lh src/
total 0
-rw-r--r-- 1 artem artem 0 Oct  2 00:29 0
-rw-r--r-- 1 artem artem 0 Oct  2 00:29 1
-rw-r--r-- 1 artem artem 0 Oct  2 00:29 2

artem@dev:~/testing/bind-mount$ ls -lh dst/
total 0

artem@dev:~/testing/bind-mount$ sudo mount --bind src/ dst/
artem@dev:~/testing/bind-mount$ ls -lh dst/
total 0
-rw-r--r-- 1 artem artem 0 Oct  2 00:29 0
-rw-r--r-- 1 artem artem 0 Oct  2 00:29 1
-rw-r--r-- 1 artem artem 0 Oct  2 00:29 2
```

Bind mounts привносят много нетривиальных деталей:

- bind-mount можно делать на файлы
- <http://lwn.net/Articles/689856/>



Как уместить много разделов и ФС в одном дереве каталогов: точки монтирования

Все ФС, которые надо сделать видимыми пользовательским приложениям, «подсоединяются» к уже существующим каталогам в ФС, видной пользователю.

Точка монтирования с точки зрения ядра ОС – это отметка на каталоге «начиная отсюда, поиск имени делается от корня такой-то ФС».

```
$ ls -lh ~/testing/mount/
total 0

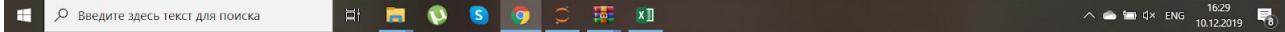
$ mount -t ext4 ~/testing/fs-images/rpmbuild ~/testing/mount/

$ ls -lh ~/testing/mount/
total 8.0K
drwxrwxr-x 1 1002 1002 4.0K Sep 25 16:59 pstorage-fes
drwxr-xr-x 1 1002 1002    83 Sep  6 21:11 rpmbuild
```

Посмотреть список точек монтирования можно так:

- `$ cat /proc/self/mounts`

Монтировать ФС можно по требованию: <https://linux.die.net/man/5/auto.master>



6.

http://heap.allinux.org/tmp/unix_base/ch02s02.html Тут начиная с заголовка “Права доступа”

Дескриптор файла обозначает открытый файл в определенном процессе. Ядро поддерживает таблицу дескрипторов файлов для каждого процесса. Каждая запись в таблице дескриптора файла указывает, что делать, если процесс запрашивает чтение, запись и другие операции над файловым дескриптором

7.

Основы построения файловых систем

Синхронный и асинхронный ввод-вывод

Диск, если начал операцию, не прерывает её до тех пор, пока она не завершится.

API для работы с файлами сохранили это же свойство – они не отдают управление, пока не завершатся.

```
for (;;) {
    int r = read(fd_src, buf, sizeof(buf));
    write(fd_dst, buf, sizeof(buf));
}
```

операция

`int r = read(fd_src, buf, sizeof(buf));
write(fd_dst, buf, sizeof(buf));`

src disk

active

`int r = read(fd_src, buf, sizeof(buf));
write(fd_dst, buf, sizeof(buf));`

idle

dst disk

idle

active

`int r = read(fd_src, buf, sizeof(buf));
write(fd_dst, buf, sizeof(buf));`

active

idle

`int r = read(fd_src, buf, sizeof(buf));
write(fd_dst, buf, sizeof(buf));`

idle

active

время

Основы построения файловых систем

Синхронный и асинхронный ввод-вывод

Диск, если начал операцию, не прерывает её до тех пор, пока она не завершится.

API для работы с файлами сохранили это же свойство – они не отдают управление, пока не завершатся.

```
for (;;) {
    int r = read(fd_src, buf, sizeof(buf));
    write(fd_dst, buf, sizeof(buf));
}
```

операция

`int r = read(fd_src, buf, sizeof(buf));
write(fd_dst, buf, sizeof(buf));`

src disk

active

active

dst disk

idle

active

`int r = read(fd_src, buf, sizeof(buf));
write(fd_dst, buf, sizeof(buf));`

idle

active

время

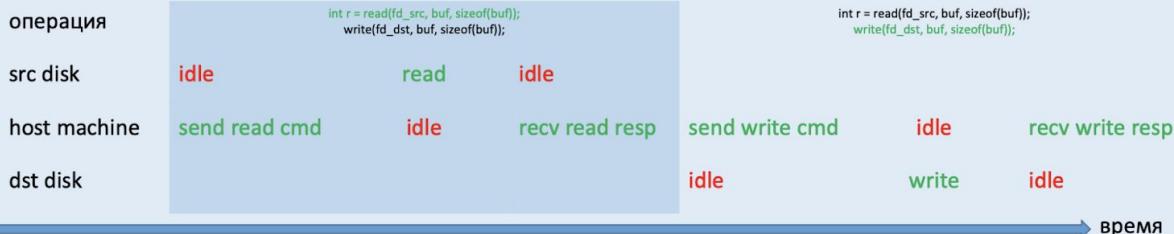
Решение: запросы на запись надо отправлять асинхронно и тут же переходить к чтению следующего блока.

Основы построения файловых систем

Pipelining и multiplexing

Также надо учитывать время, которое требуется для того, чтобы отослать команду на чтение или запись. Оно было пренебрежимо мало для HDD, но оно велико для сетей и для SSD и NVMe.

```
for (;;) {
    int r = read(fd_src, buf, sizeof(buf));
    write(fd_dst, buf, sizeof(buf));
}
```



Решение: запросы на чтение надо отправлять в таком количестве, чтобы у диска всегда была непустая очередь команд. Первая команда всё равно увидит задержку на отправку запроса и получение ответа, но для последующих этой задержки не будет.

Основы построения файловых систем

Pipelining и head-of-line blocking

Предположим, что мы послали много запросов к диску (или к серверу). В каком порядке будут отсыпаться ответы?

Есть два возможных варианта:

- в порядке получения запросов,
- в порядке завершения.

Первый вариант (pipelining) зачастую можно реализовать для протоколов, где изначально не позаботились о мультиплексировании.

Второй вариант требует поддержки в протоколе: у запросов должны быть уникальные номера.

Pipelining имеет существенный недостаток: если серверу были отправлены запросы R₁, R₂, ..., то R₂ и последующие должны ждать, пока закончится R₁. Если он окажется очень медленным, то все следующие за ним проведут много времени в очереди, даже если бы могли исполниться быстро. Такое явление называется head-of-line blocking.

11.

Основы построения файловых систем

Два способа записать целое число в память или на диск

В начале идут старшие байты (**Big-endian**)

u32 x = 0xA2B3C4D;

На диске:

1A 2B 3C 4D |

В начале идут младшие байты (**little-endian**)

u32 x = 0xA2B3C4D;

На диске:

4D 3C 2B 1A |

12.

Все блоки одного файла ФС старается сохранить в одной блок-группе - нужно для локальности данных, головке не надо много перемещаться.

Acronis @ МФТИ

Основы построения файловых систем

Устройство ext2 в целом

The diagram illustrates the layout of an ext2 file system. It starts with a 'Пустое место длиной 1Kb' (empty 1Kb space). Following this are two 'Block group 0' and 'Block group 1' structures, which are repeated for each subsequent block group. Each block group contains the following components in sequence from left to right: SB (Superblock), BG headers, block bitmap, inode bitmap, inode table, and data. A blue arrow at the bottom indicates the direction 'от младших адресов к старшим' (from younger addresses to older ones).

Superblock (SB) содержит информацию о файловой системе в целом: её размер, размер и число блоков и т.п.

Block Group Header содержит информацию об отдельной группе блоков: число свободных блоков и инод.

Block bitmap – это битовый массив, определяющий, которые из блоков заняты, а какие свободны. Блок – это минимальная единица выделения места на ФС.

Inode bitmap – это битовый массив, определяющий, который из инод заняты, а какие свободны. Inode (Index node) – это структура, описывающая один файл на ext2.

Inode table – это область на диске, хранящая содержимое инод. Они расположены как непрерывный массив.

1 индирект блок указывает на 1024 директ блока 4к(блок)/32Б(указатель)
Минусы - чтобы найти индирект блок надо передвинуть головку на иноду

Index nodes (src/linux/fs/ext2/ext2.h)

В ext2 информация о свойствах файла и его расположении на диске содержится в структуре index node:

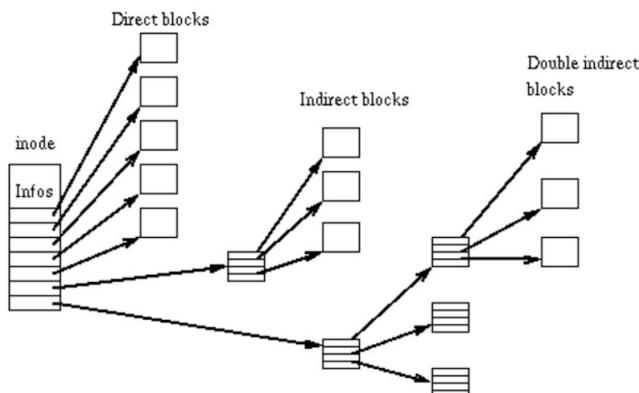
```
struct ext2_inode {
    __le16 i_mode;          /* File mode */
    __le16 i_uid;           /* Low 16 bits of Owner Uid */
    __le32 i_size;          /* Size in bytes */
    __le32 i_atime;         /* Access time */
    __le32 i_ctime;         /* Creation time */
    __le32 i_mtime;         /* Modification time */
    __le32 i_dtime;         /* Deletion Time */
    __le16 i_gid;           /* Low 16 bits of Group Id */
    __le16 i_links_count;   /* Links count */
    __le32 i_blocks;         /* Blocks count */
    __le32 i_flags;          /* File flags */
    __le32 i_osd1;
    __le32 i_block[EXT2_N_BLOCKS];/* Pointers to blocks */
    __le32 i_generation;    /* File version (for NFS) */
    __le32 i_file_acl;      /* File ACL */
    __le32 i_dir_acl;       /* Directory ACL */
    __le32 i_faddr;          /* Fragment address */
    __le8 i_osd2[12];
};
```

Основы построения файловых систем

Index nodes (src/linux/fs/ext2/ext2.h)

В ext2_inode->i_block[] хранится список блоков, которые составляют файл. Но в этом массиве 15 элементов. Как быть с файлами, которые длиннее 15 блоков?

Последние три элемента в i_block[] косвенные, т.е. указывают на блоки, которые сами являются списками блоков. Они имеют уровни косвенности 1, 2 и 3, соответственно.

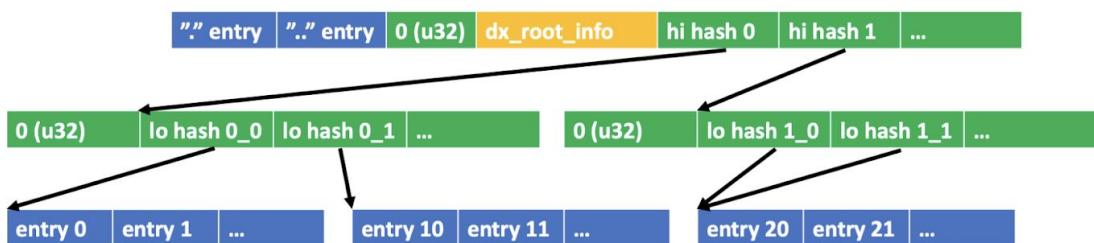


13.

Для поиска файла по имени в РНTree-ext3 находим хэш для файла, делим его пополам, ищем верхнюю часть в списке верхних хэшей, переходим в соответствующий блок с нижними, ищем там нижний, а потом попадаем в список из директорий - такой же реализован в ext2.

Каталоги в ext3 (hash indexed dirs)

Для больших каталогов используется следующее представление*:



- Если много имён имеют совпадающий хеш и их список не умещается в один блок, то в карте "хеш → номер блока" ставится флаг «список продолжается в следующем блоке».
- Разные хеши могут ссылаться на один блок.

Изображённые выше блоки на диске располагаются подряд (и составляют один файл).

Нулевые записи в блоках нижнего уровня и нулевое 4-байтовое значение в корневом блоке поставлены затем, чтобы алгоритм линейного поиска из ext2 увидел правильный список элементов (вспоминаем, что элемент с нулевым полем inode – это признак «в этом блоке больше нет записей»).

* Каждый узел изображённого дерева занимает один блок на диске.

Фичи

Superblock

Пустое место длиной 1Кб	Block group 0						Block group 1						...
	SB	BG headers	block bitmap	inode bitmap	inode table	data	SB	BG headers	block bitmap	inode bitmap	inode table	data	
<pre>struct ext2_super_block { __le32 s_inodes_count; /* Inodes count */ __le32 s_blocks_count; /* Blocks count */ __le32 s_r_blocks_count; /* Reserved blocks count */ __le32 s_free_blocks_count; /* Free blocks count */ __le32 s_free_inodes_count; /* Free inodes count */ __le32 s_first_data_block; /* First Data Block */ __le32 s_log_block_size; /* Block size */ __le32 s_log_frag_size; /* Fragment size */ __le32 s_blocks_per_group; /* # Blocks per group */ __le32 s_frags_per_group; /* # Fragments per group */ __le32 s_inodes_per_group; /* # Inodes per group */ __le32 s_mtime; /* Mount time */ __le32 s_wtime; /* Write time */ __le16 s_mnt_count; /* Mount count */ __le16 s_max_mnt_count; /* Maximal mount count */ __le16 s_magic; /* Magic signature */ __le16 s_state; /* File system state */ __le16 s_errors; /* Errors */ __le16 s_minor_rev_level; /* minor revision level */ __le32 s_lastcheck; /* time of last check */ __le32 s_checkinterval; /* max. time between checks */ __le32 s_creator_os; /* OS */ __le32 s_rev_level; /* Revision level */ __le16 s_def_resuid; /* Default uid for reserved blocks */ __le16 s_def_resgid; /* Default gid for reserved blocks */ __le32 s_first_ino; /* First non-reserved inode */ __le16 s_inode_size; /* size of inode structure */ __le16 s_block_group_nr; /* block group # of this sb */ __le32 s_feature_compat; /* compatible features */ __le32 s_feature_incompat; /* incompatible features */ __le32 s_feature_ro_compat; /* readonly-compatible features */ __u8 s_uuid[16]; /* 128-bit uuid for volume */ char s_volume_name[16]; /* volume name */ char s_last_mounted[64]; /* directory where last mounted */ __le32 s_algorithm_usage_bitmap; /* For compression */</pre>													

Основы построения файловых систем

Compat, ro-compat, incompat features

Compat features: старые реализации ext2 могут читать, и писать на такую файловую систему.

- EXT4_FEATURE_COMPAT_DIR_PREALLOC
- EXT4_FEATURE_COMPAT_HAS_JOURNAL
- EXT4_FEATURE_COMPAT_EXT_ATTR
- EXT4_FEATURE_COMPAT_DIR_INDEX (hash directories)

Ro-compat features: старые реализации могут корректно читать такую ФС, но писать в неё уже нет.

- EXT4_FEATURE_RO_COMPAT_SPARSE_SUPER
- EXT4_FEATURE_RO_COMPAT_HUGE_FILE
- EXT4_FEATURE_RO_COMPAT_QUOTA

Incompat features: старые реализации не могут смонтировать такую ФС.

- EXT4_FEATURE_INCOMPAT_COMPRESSION
- EXT4_FEATURE_INCOMPAT_JOURNAL_DEV
- EXT4_FEATURE_INCOMPAT_EXTENTS
- EXT4_FEATURE_INCOMPAT_INLINE_DATA
- EXT4_FEATURE_INCOMPAT_ENCRYPT

Compat, ro-compat, incompat features

Compat features: старые реализации ext2 могут читать, и писать на такую файловую систему.

RO-compat features: старые реализации могут корректно читать такую ФС, но писать в неё уже нет.

Incompat features: старые реализации не могут смонтировать такую ФС.

Compat-discard features (QCOW2): старые реализации могут читать, и писать, но должны обнулить указатели на структуры, которые они не поддерживают.

Пример: CBT map (Changed Block Tracking map).

14.

Основы построения файловых систем

Создание файла и исчезновение питания

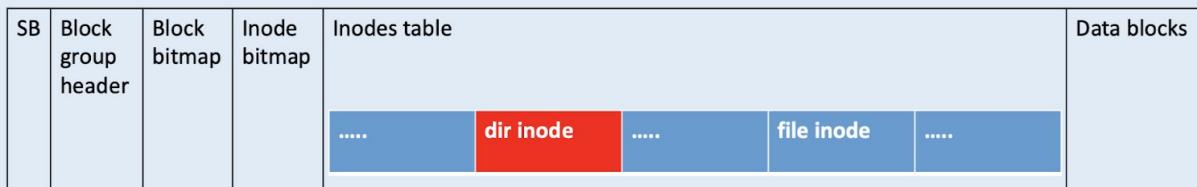
С точки зрения пользователя всё просто:

```
int fd = open("fes.c", O_RDWR|O_CREAT|O_EXCL, S_IRUSR|S_IWUSR);
```

С точки зрения ФС (для примера возьмём ext2) действий больше:

1. Отыскать незанятую inode для файла,
2. Отыскать свободный блок для содержимого файла,
3. Пометить найденные inode и блок как используемые,
4. Заполнить inode для файла,
5. Записать struct ext2_dir_entry в каталог, где создан файл,
6. Из-за этой записи длина каталога подрастёт, поэтому надо обновить inode для каталога.

Области на диске:



Acronis @ МФТИ

Основы построения файловых систем

Создание файла и исчезновение питания

С точки зрения ФС (для примера возьмём ext2) действий больше:

1. Отыскать незанятую inode для файла,
2. Отыскать свободный блок для содержимого файла,
3. Пометить найденные inode и блок как используемые,
4. Заполнить inode для файла,



Тут выключилось питание



5. Записать struct ext2_dir_entry в каталог, где создан файл,
6. Из-за этой записи длина каталога подрастёт, поэтому надо обновить inode для каталога.

В итоге имеем:

1. Блок и inode отмечены как занятые,
2. Счётчики свободных блоков и inode уменьшены,
3. Файла нет.

Можно изменить порядок записи, но тогда при прерывании между 4 и 5 также будет ошибка, поэтому можно fsync'ать

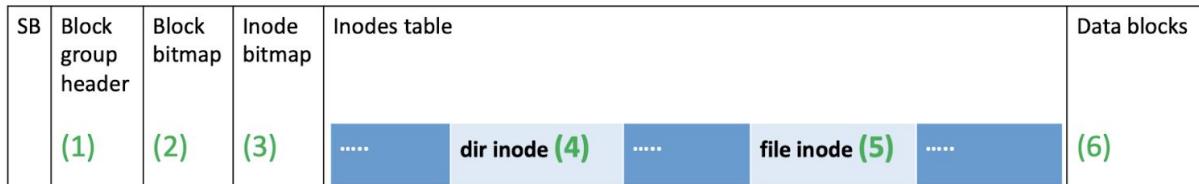
Acronis @ МФТИ

Основы построения файловых систем

Что должен обеспечить журнал на ФС

1. Упорядочивание изменений в ФС,
2. Транзакционность изменений в ФС.

Если при создании файла изменения сделать видимыми в таком порядке:



Размер каталога подрос до того, как в нём появилась новая запись – читатель увидит мусор.

Теоретически, упорядочивания можно добиться, если писать блоки в нужном порядке, и каждый раз делать fsync() после записи. Но так будет слишком медленно.

Короче всё это бред, вводим журнал!

Основы построения файловых систем

Что должен обеспечить журнал на ФС

1. Упорядочивание изменений в ФС,
2. Транзакционность изменений в ФС.

Реализация:

1. Записываем блоки, подлежащие изменению, в журнал,
2. fsync(),
3. Асинхронно меняем состояние диска.
4. Когда закончили изменять состояние диска, делаем запись в журнале о том, что транзакция применена.

Журнал						Содержимое диска						
hdr	0	1	2	0	...	2	...	1	...

Если при обновлении диска произошёл сбой (отключение питания или падение ОС), то при следующем монтировании ФС мы можем применить изменения, написанные в журнале, и доделать изменения, которые не применили из-за падения.

Эта процедура называется **crash recovery**.

Основы построения файловых систем

Что журналировать? Consistent FS state.

Журналирование, предложенное на прошлом слайде, удваивает число блоков, которые надо записать на диск.

Это не так плохо для скорости: запись в журнал последовательная.

Но всё равно хочется журналировать поменьше данных.

Идея:

- Будем журналировать только метаданные ФС

В результате, после crash recovery мы будем получать неразломанную ФС: без потерянных блоков и инод, без dir_entry, ведущих в никуда, etc.

Но: ФС не предоставляет **никаких** гарантий о том, что будет с пользовательскими данными.

В файловых системах ext3/ext4 есть такие режимы журналирования (указываются при монтировании):

- data=writeback (журналируются только метаданные, пользовательские данные записываются только в блоки с данными)
- data=ordered (журналируются только метаданные, но только после того, как соответствующие пользовательские данные записаны на диск)
- data=journal (журналируются и метаданные, и данные)

Вопрос: при каких из этих режимах гарантируется консистентность метаданных файловой системы? А пользовательских данных?

Журналирование лишь дает нам уверенность, что после выключения все метаданные сохранятся. Например размер файла точно не изменится, но сам файл может быть покарапчен.

Основы построения файловых систем

Что писать в журнал?

Логические изменения в ФС:

- Добавление/удаление/переименование файлов,
- Изменение размера файлов,
- Изменение атрибутов,
- ...

Физические изменения: содержимое блоков ФС, которое должно получиться после применения транзакции.

Что будет, если во время проигрывания журнала исчезнет питание и надо будет повторить проигрывание журнала?

Дважды проиграть транзакции в общем случае нельзя:
как повторить rename("a", "b")?

Операции “записать такое-то содержимое поверх блока с номером N” **идемпотентны**: их можно повторять много раз с тем же эффектом, который даёт однократное повторение.

15.

Основы построения файловых систем

Что журналировать? Consistent FS state.

Рассмотрим пример добавления данных в конец файла на XFS.

XFS

- Найдёт свободные нулевые блоки,
- Зажурнилирует обновление block bitmap, inode, и extent tree,
- Даст пользовательскому приложению писать в выделенные блоки.

В журнал попадут только изменения метаданных ФС (килобайты). Запись же пользовательских данных (потенциально – гигабайты) пойдёт мимо журнала.

Что произойдёт при падении ОС во время такой записи в файл?

Получится подросший в размере файл, в хвосте которого будет мусор.

Ещё причина выбрать такое поведение: ошибки отложенного writeback.

<https://lwn.net/Articles/457667/>

Режим - writeback, поэтому такая ошибка.

16.

Идемпотентность (лат. *idem* — тот же самый + *potens* — способный) — свойство объекта или операции при повторном применении операции к объекту давать тот же результат, что и при первом. Термин предложил американский математик Бенджамин Пирс (англ. *Benjamin Peirce*) в статьях 1870-х годов.

Сказать, про операцию rename из 14 вопроса.

Основы построения файловых систем

Case study: идемпотентность операций

Рассмотрим протокол Acronis для общения со стораджем для бекапов:

- Open: file_name, lock_level --> lock_id,
- Read: file_name, lock_id, offset, size --> data,
- Append: file_name, lock_id, data,
- PunchHole: file_name, lock_id, hole,
- Close: lock_id,
- Rename: file_path_src, file_path_dst.

Возможные значения lock_id:

- Shared,
- Normal (может быть только 1, разрешает другие shared),
- Exclusive (может быть только 1, запрещает любые другие блокировки).

Какие тут есть проблемы? Подсказка: запросы передаются по сети.

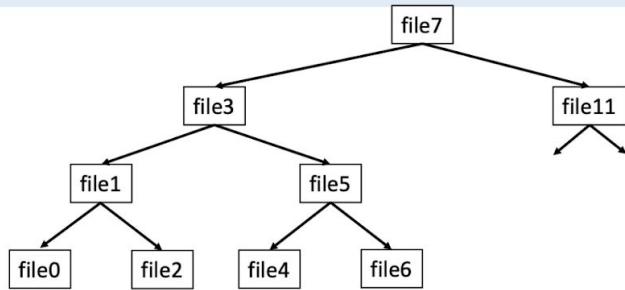
Ответ: open, close, rename не идемпотентны и требуют, чтобы сетевые соединения никогда не рвались.

Напоминание: как организовать список файлов?

Линейный список, где файлы идут в порядке создания

Дерево поиска

file15, file1,
file2, file3,
file4, file9,
file6, file8,
file7, file5,
file12, file11,
file10, file13,
file14, file0



Переход на начало списка:

 $\approx 10\text{msec}$

Чтение списка:

 $\approx 1\text{msec} (\approx 1\text{MB})$

Поиск (список уместился в RAM):

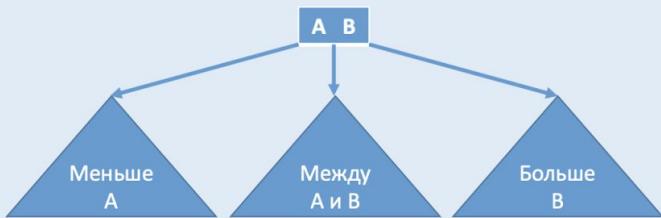
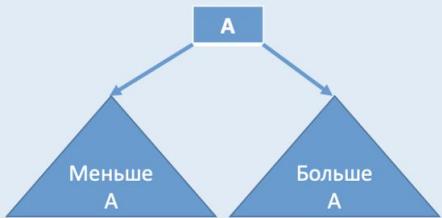
 $<< 1\text{msec}$

Тут нужны 4 позиционирования читающей головки, т.е.
меньше, чем в 40msec мы не уложимся.

2-3-деревья и красно-чёрные деревья (напоминание)

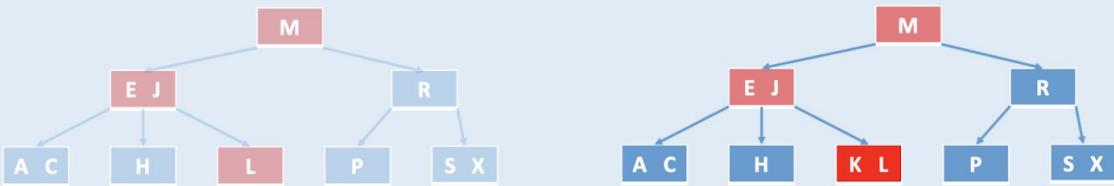
2-3-дерево – один из способов хранить множество элементов. Оно определяется следующими свойствами:

- каждый узел содержит один или два элемента из множества,
- узлы имеют 0, 2 или 3 потомка,
- дерево идеально сбалансировано,
- значения элементов в узлах упорядочены:

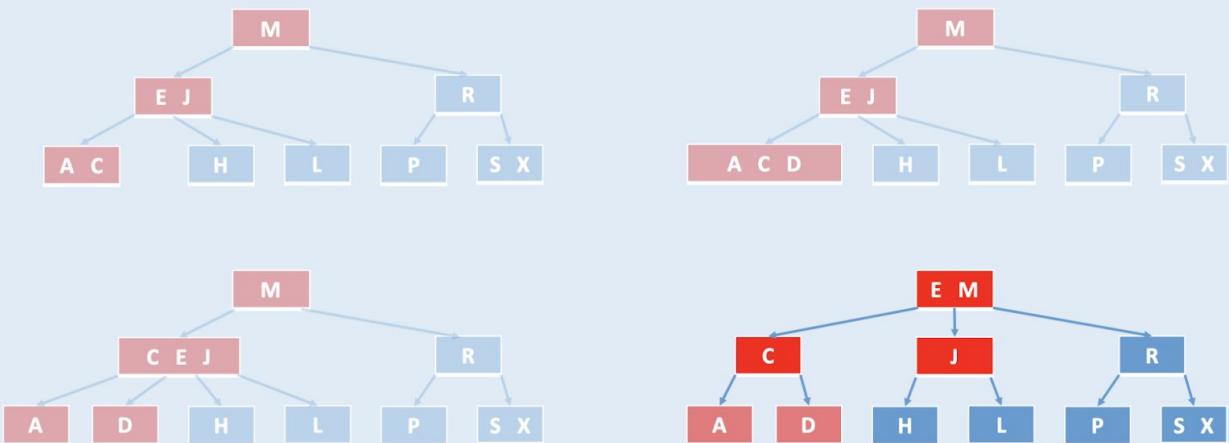


2-3-деревья и красно-чёрные деревья (напоминание)

Вставка в узел с одним элементом:

**2-3-деревья и красно-чёрные деревья (напоминание)**

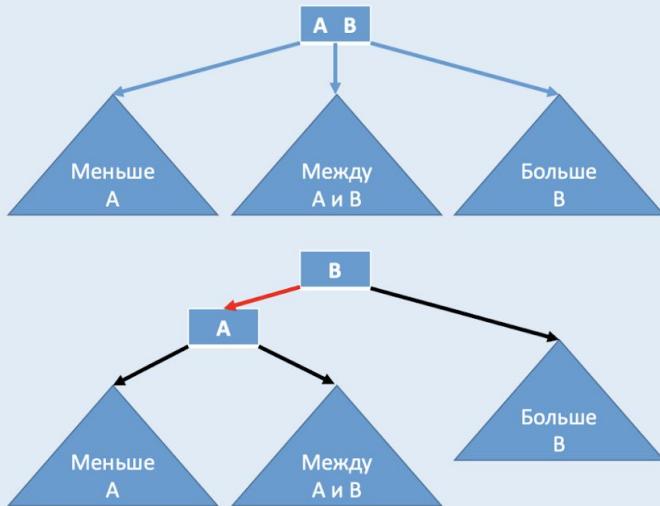
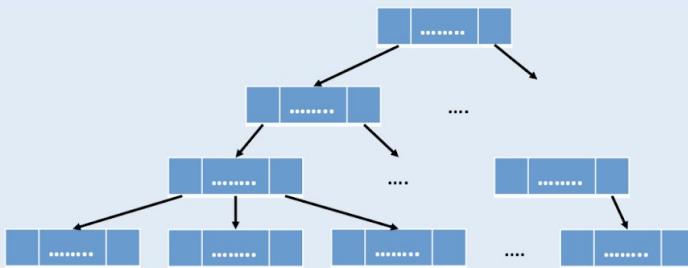
Вставка в узел с двумя элементами:



2-3-деревья и красно-чёрные деревья (напоминание)

2-3-деревья взаимно-однозначно соответствуют красно-чёрным:

В деталях рассказано здесь:

<https://algs4.cs.princeton.edu/33balanced/>**18.****В-деревья**

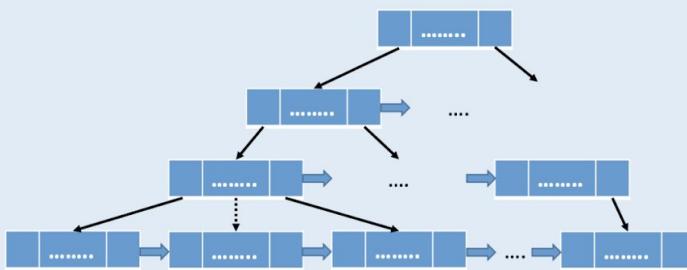
- В узлах хранятся массивы пар (k_i, p_i) , упорядоченные по возрастанию ключей,
- Массивы имеют ограниченную длину: от L до $2L$ элементов,
- Указатели на страницы данных хранятся в листьях, во внутренних узлах – ссылки на страницы с узлами-потомками,
- Все листья расположены на одной глубине,
- Указатель p_i ссылается на поддерево с ключами в диапазоне $[k_i, k_{i+1})$ (внимание: $k_{m+1}!$),
- Если при вставке происходит переполнение узла, то он разделяется на два узла длины L , а средний элемент перемещается в родительский узел.

Есть проблемы:

- Вставки и удаления создают случайное IO,
- Удаление зачастую реализуется нетривиально, или оставляет много мусора,
- В многопоточной среде надо брать блокировки сразу на весь путь до листа.

Проблема возникает, когда один ищет элемент из самого левого листа, а второй вставляет элемент в самый левый лист (с переполнением). Тогда если искомый элемент расщепится в правую часть, то сёрчер его не найдёт.

B^{link}-деревья (Lehman, Yao)*



При расщеплении узла не обязательно модифицировать родителя – хватит проставить ссылку на правого соседа, а родительский узел можно модифицировать потом.

В итоге, в каждый момент времени достаточно держать блокировку только на одном узле.

<https://www.csd.uoc.gr/~hy460/pdf/p650-lehman.pdf>

* Используются, например, в PostgreSQL.

Acronis @ МФТИ

Тут такой проблемы не будет.

19.

Основы построения файловых систем

Слияние двух B-деревьев

Заметим, что два множества, представленные B-деревьями, легко объединить за линейное время:

1. итерирование по листьям даёт элементы в порядке возрастания ключей; отсортированные списки ключей из двух деревьев можно объединить в один, как в сортировке слиянием,
2. отсортированный объединённый список выписываем в страницы, расположенные последовательно,
3. Для каждого 2L-1 подряд идущих страниц-узлов делаем директорную страницу; разные директорные страницы тоже выписываем последовательно,
4. Аналогично создаём директорные страницы более высокого уровня,
5. Повторяем до тех пор, пока не напишем одну директорную страницу, которую называем корнем объединённого дерева.

Важные свойства:

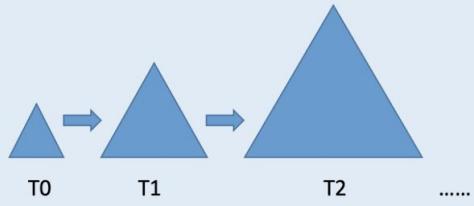
- Обход листьев объединяемых деревьев генерирует преимущественно линейное чтение,
- Страницы-листы объединённого дерева выписываются линейно.

Слияние B-tree быстрое, так как генерирует последовательное IO.

Log-Structured Merge Tree

LSM-дерево – это иерархия B-деревьев.

- Поиск элемента делается по очереди в деревьях T_0, T_1, \dots
- Вставки делаются только в дерево T_0 ,
- Дерево T_0 (возможно, несколько первых) располагается в RAM, гарантируя быструю вставку,
- При переполнении дерева T_i оно сливается с деревом T_{i+1} ; полученное дерево объявляется новым T_{i+1} ,
- Удаление элемента реализуется как вставка элемента, помеченного флагом «удалённый»*. Фактическое удаление произойдёт при слиянии деревьев.



Факт: оптимальная производительность вставок в LSM-дерево (наименьшие накладные расходы на слияния) достигается, если число элементов в деревьях T_0, T_1, \dots образует геометрическую прогрессию.

<http://citeseervx.ist.psu.edu/viewdoc/download?doi=10.1.1.44.2782&rep=rep1&type=pdf>
<http://www.benstopford.com/2015/02/14/log-structured-merge-trees/>

То есть мы только слияем деревья не в памяти, а на диске, а слияние – быстрая операция.

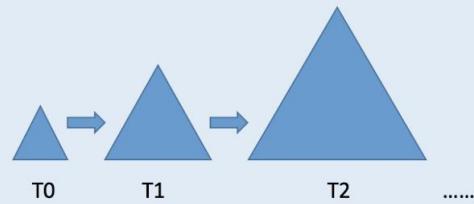
Log-Structured Merge Tree

При таком подходе вставки и удаления почти всегда получаются очень быстрыми, поскольку работают только с деревом в RAM.

Но теперь необходимо иногда выполнять слияние деревьев. Как мы обсудили, это можно сделать быстро и генерировать только эффективные последовательности чтений и записей.

Остаются две большие проблемы:

- Поиск необходимо выполнять не в одном дереве, а во многих.
- Write amplification и непредсказуемые задержки: у LSM-дерева мало амортизированное время вставки в дерево, но вставки, которые приводят к слиянию деревьев, будут исполняться на несколько порядков дольше, чем типичные вставки, и создадут много IO.



Bloom filters

Поиск в LSM-дереве приходится реализовывать как несколько поисков по его составляющим разных уровней.

Можно избежать поиска во многих деревьях T_i , если научиться быстро определять, что искомого ключа в T_i не содержится. Это делает фильтр Блума, вероятностная структура данных, которая по множеству и ключу может выдавать ответы

- элемента в множестве нет,
- элемент в множестве может присутствовать.

Конструкция фильтра Блума: пусть имеется битовый массив длиной m и k независимых хеш-функций f_i , принимающих значения в диапазоне $[0, m-1]$.

- При вставке элемента x установим в 1 биты, стоящие на местах $f_1(x), f_2(x), \dots, f_k(x)$,
- Для проверки отсутствия элемента y проверим, установлены ли биты на позициях $f_1(y), f_2(y), \dots, f_k(y)$.

Если элементы x берутся из множества мощностью N и вероятность неправильного ответа «может присутствовать» не должна превышать p , то для построения фильтра надо взять

$$k \geq -\log_2(p)$$

хеш-функций и битовый массив длины

$$m \geq k * N / \ln(2)$$

20.

Power of two choices

Зачем в фильтре Блума использовать несколько независимых хешей?

Факт 1: Пусть дана хеш-таблица, где элементы размещаются по хешу в N списков. Вставим в неё N случайных элементов. Какова будет длина максимально заполненного списка? С вероятностью $\geq 1 - O(1/N)$ она будет равна

$$\frac{\log N}{\log \log N} + O(1)$$

Факт 2: Пусть дана хеш-таблица, где элементы размещаются в N списках, но правило размещения таково: при вставке элемента посчитаем для него d независимых хешей и выберем самый короткий список, соответствующий одному из полученных хешей.

Вставим N случайных элементов в такую хеш-таблицу. Какова будет длина максимально заполненного списка на этот раз? С вероятностью $\geq 1 - O(1/N)$ она составит

$$\frac{\log \log N}{\log d} + O(1)$$

Использование двух хеш-функций вместо одной улучшает асимптотику длины наиболее занятого списка. Использование большего числа только уменьшает константу в $O()$.

Power of two choices

Применения:

- хеш-таблицы,
- фильтры Блума,
- балансировка нагрузки в распределённых системах,
- уменьшение tail latency в распределённых системах.

Пример: если есть несколько HTTP-серверов, с которых можно скачать файл, то можно очень просто распределить нагрузку между ними:

1. выбрать два случайных сервера,
2. отправить обоим один и тот же запрос,
3. с того, кто первым начнёт слать ответ, скачать файл,
4. у более медленного отменить запрос.

Проблема: удвоение числа запросов (не считая запросов на отмену).

https://people.cs.umass.edu/~ramesh/Site/PUBLICATIONS_files/MRS01.pdf

Power of two choices

Применения:

- хеш-таблицы,
- фильтры Блума,
- балансировка нагрузки в распределённых системах,
- уменьшение tail latency в распределённых системах.

Вопрос: пусть у нас есть N одинаковых серверов, которые 99% запросов обрабатывают < 10ms, а 1% запросов (случайных) обрабатывают 1s. Если для построения ответа пользователю требуется получить ответ от 10 серверов, то какая доля пользовательских запросов будет обработана за 10ms?

<http://cseweb.ucsd.edu/~gmporter/classes/fa17/cse124/post/schedule/p74-dean.pdf>

Power of two choices

Применения:

- хеш-таблицы,
- фильтры Блума,
- балансировка нагрузки в распределённых системах,
- уменьшение tail latency в распределённых системах.

Идея: послать запрос одному серверу и, если время ответа превысило 90-й (95-й) перцентиль, то перепослать запрос уже другому серверу.

Заодно это решает проблему удвоением числа запросов при наивном применении power of two choices.

<http://cseweb.ucsd.edu/~gporter/classes/fa17/cse124/post/schedule/p74-dean.pdf>

21.

Рассказать про эксперимент CERNa про надёжность HHD. Есть методы проверки целостности данных.

Основы построения файловых систем

Способы проверки целостности данных

У нас упоминались два инструмента для проверки целостности данных:

- Cyclic redundancy checks,
- Криптографические хеш-суммы.

Обсудим их детальнее.

Основы построения файловых систем

Cyclic Redundancy Check

Рассмотрим сообщение как последовательность битов (элементов GF(2)) и сопоставим ему многочлен из $GF(2)[X]$:

$$a_{n-1}a_{n-2} \dots a_0 \leftrightarrow M(X) = X^n + a_{n-1}X^{n-1} + a_{n-2}X^{n-2} + \dots + a_0$$

Возьмём многочлен $C \in GF(2)[X]$ степени d , посчитаем $r(X)$ – остаток от деления $M(X) * X^d$ на $C(X)$.

$r(X)$ называется CRC сообщения M .

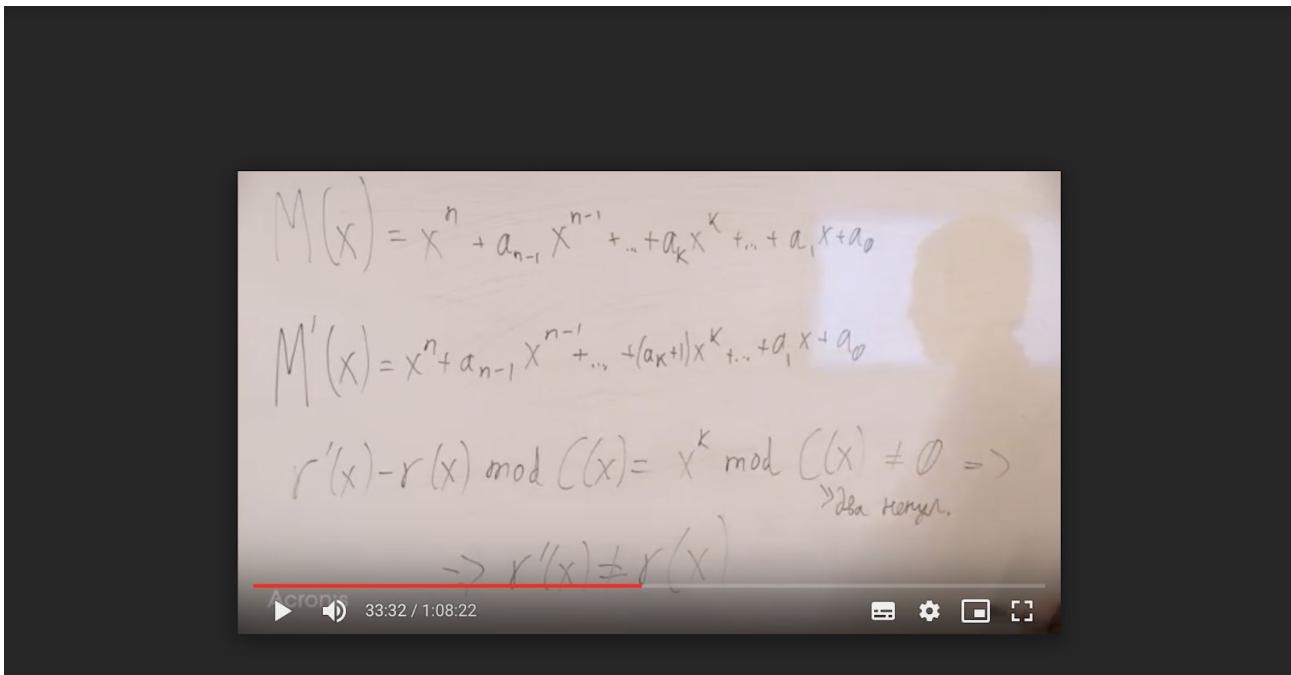
Теперь построим многочлен

$$M(X) * X^d + r(X)$$

Ему соответствует сообщение $a_{n-1}a_{n-2} \dots a_0$, к которому дописали биты, равные коэффициентам r (внимание: r может быть степени меньше m , тогда считаем коэффициенты при старших степенях нулями).

CRC очень хорошо приспособлены для аппаратной реализации: из арифметических операций нужен только XOR.

Многочлен $C(X)$ подбирается так, чтобы обеспечить обнаружение определённых типов ошибок.



Основы построения файловых систем

Ошибки, которые находят CRC (упражнения)

- Если $C(X)$ имеет два и более ненулевых коэффициентов, то он определяет любую ошибку, изменяющую только один бит.
- Если в разложении $C(X)$ на неприводимые множители есть многочлен степени m , то $C(X)$ определяет любую ошибку, изменяющую только два бита, расположенных на расстоянии, меньшем m .
- Если $C(X)$ делится на $X+1$, то он определяет любую ошибку, меняющую нечетное число бит.

Cyclic Redundancy Check

Типичная схема применения CRC:

```
struct something
{
    some fields
    ...
    u64 crc;
}
```

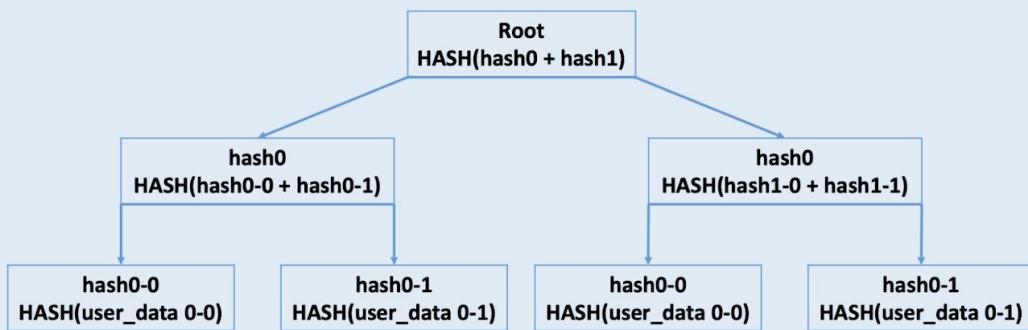
1. Вычислить CRC всех полей структуры, кроме `something->crc`,
2. Записать в `something->crc` такое значение, чтобы CRC от всей структуры равнялся нулю.

Упражнение: пусть дано сообщение M и порождающий многочлен $C(X)$ степени d . Найти целое d -битовое число X такое, что $\text{CRC}(\text{concat}(M, X)) = 0$.

CRC - быстрее криптографических кэшей, но CRC легко обмануть, если покарантить определенные биты.

Иногда нам не надо проверять на целостность все данные, которые есть. А надо проверить часть данных.

Пример проверки целостности дерева: Merkle trees



Применения:

- проверка целостности структуры дерева каталогов и дерева екстентов (ZFS, btrfs),
- проверка подлинности данных в p2p-сетях,
- быстрое определение частей деревьев, подлежащих синхронизации в распределённой БД (например, DynamoDB).

Если в каком-то узле не сходится, значит в поддереве локализована ошибка. Идем снизу вверх и сравниваем кэши. Например хотим проверить 00, от него считаем и кэш сравниваем с сущ, потом вычисляем верхний уровень

используя известный 01 и опять сверяем, ...

Acronis @ МФТИ

Основы построения файловых систем

Эксперимент в CERN о надёжности хранения данных

Выводы:

- Данные нельзя хранить в единственном экземпляре,
- Необходимы контрольные суммы для проверки целостности,
- Необходима активная фоновая проверка данных.
- Хранение реплик или использование Reed-Solomon,
- ZFS и btrfs хранят криптографические хеши всех записанных данных, ext4 хранит только CRC,
- Online scrubbing & repair в ZFS и btrfs или в HW RAID-контроллерах.

22.

Берём много недорогих дисков и делаем что-то с ними.

Acronis @ МФТИ

Основы построения файловых систем

RAID – Redundant Array of Independent (Inexpensive) Disks

Для чего нужен:

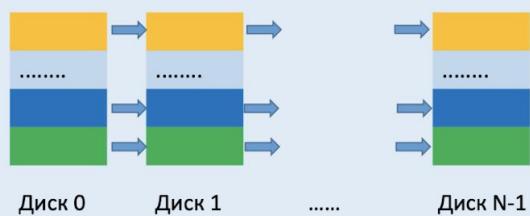
- Большая надёжность, чем у отдельных дисков,
- Большая вместимость, чем у отдельных дисков.

Основы построения файловых систем

Уровни RAID

RAID0 (stripe)

Данные разрезаются на последовательные куски длины $N * B$, каждый кусок разделяется на N частей, которые записываются на различные диски:

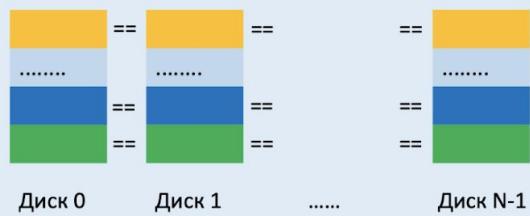


Основы построения файловых систем

Уровни RAID

RAID1 (mirror)

Каждый диск в массиве содержит одни и те же данные:



Уровни RAID

RAID4

Массив состоит из $N+1$ дисков. На первых N дисках данные хранятся, как на RAID0. На последнем диске каждый блок вычисляется как XOR соответствующих блоков на N дисках.

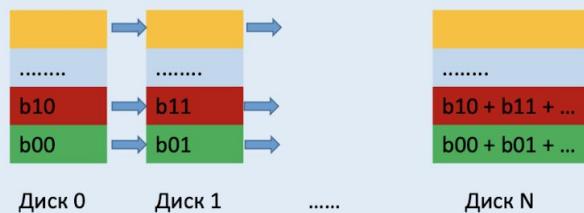


При потере любого диска массив остаётся работоспособным.

Уровни RAID

RAID4

Массив состоит из $N+1$ дисков. На первых N дисках данные хранятся, как на RAID0. На последнем диске каждый блок вычисляется как XOR соответствующих блоков на N дисках.



При потере любого диска массив остаётся работоспособным.

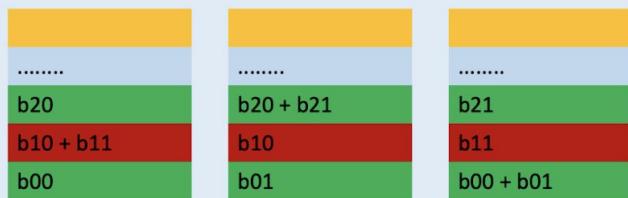
Такой массив имеет концептуальный недостаток: диск с блоками чётности будет изнашиваться быстрее других дисков.

Основы построения файловых систем

Уровни RAID

RAID5

Массив строится так же, как и RAID4, но блоки чётности в разных страйпах хранятся на разных дисках:



Acronis @ МФТИ

Основы построения файловых систем

Write holes

Запись на разные диски будет происходить в разное время.

Рассмотрим такой сценарий:

1. начинается запись на RAID1,
2. диск #0 обработал запрос на запись сектора,
3. произошёл сбой питания,
4. на диске #1 сектор остался без изменений.

Write holes

Аппаратный способ решения:

- BBU (Battery Backup Unit) в RAID-контроллерах.

Программные способы решения:

- write intent bitmap (linux md),
- checksumming + COW (ZFS),
- SSD journal: <https://lwn.net/Articles/665299/>.

Write intent bitmap, помимо исправления write holes, позволяет уменьшить время проверки и перестроения массива после аварийного выключения.

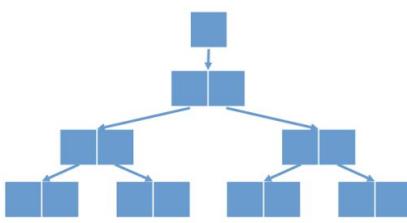
23.

Требования к ФС и различные проблемы

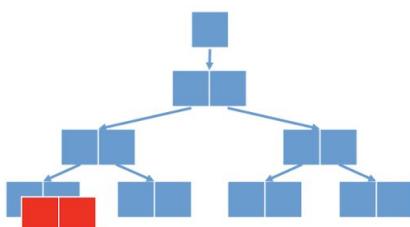
1. ФС всегда должна быть в согласованном состоянии.
2. Быстрый FSCheck или, что лучше, отсутствие оного.
3. Быстрая запись в файлы и быстрая модификация метаданных.
4. Гибкое управление размером ФС, возможность использовать несколько дисков одновременно для большей надёжности или скорости.
5. Быстрые снимки состояния ФС, клоны ФС и откат к предыдущему состоянию.
6. Защита от случайных повреждений содержимого дисков. Для RAID, защита от RAID write hole.
 - Ext4 и XFS могут возвращать мусор при чтении (ср. эксперимент о порче содержимого дисков в CERN).

Идея: copy-on-write transactions (ZFS и WAFL)

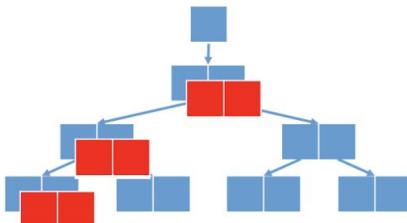
0. Исходное дерево



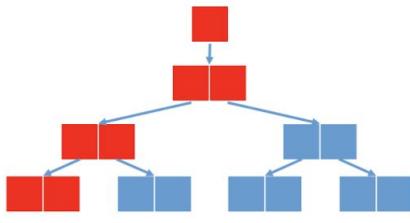
1. Создадим копии изменённых блоков



2. Создадим копии блоков выше по дереву



3. Перепишем суперблок ФС



Acronis @ МФТИ

Требования к ФС и что нам даёт Copy-on-write

ФС всегда должна быть в согласованном состоянии.	Суперблок всегда указывает на целостную ФС.
Быстрый FSCK или, что лучше, отсутствие оного.	FSCK не нужен.
Быстрая запись в файлы и быстрая модификация метаданных.	Блоки-копии можно выписывать последовательно, притом неважно, принадлежат они одному файлу или разным. Вопрос: как быть с фрагментацией файлов и производительностью чтения?
Гибкое управление размером ФС, возможность использовать несколько дисков одновременно для большей надёжности или скорости.	
Быстрые снимки состояния ФС, клоны ФС и откат к предыдущему состоянию.	Создание снимка бесплатно: надо просто не удалить старый суперблок, а сохранить ссылку на корень ФС как ссылку на корень снапшота. Откат к предыдущему состоянию тоже тривиален.
Защита от случайных повреждений содержимого дисков.	
Защита от RAID write hole.	RAID write hole возникают при перезаписи блоков, а в Copy-on-write FS перезаписи никогда не происходят.

Acronis @ МФТИ

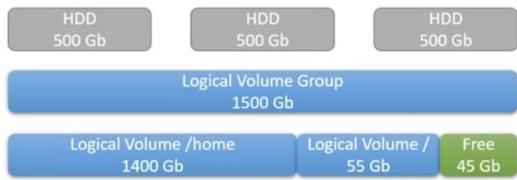
Быстрое чтение происходит из-за того, что верхние блоки лежат в page-cache

Основы построения файловых систем

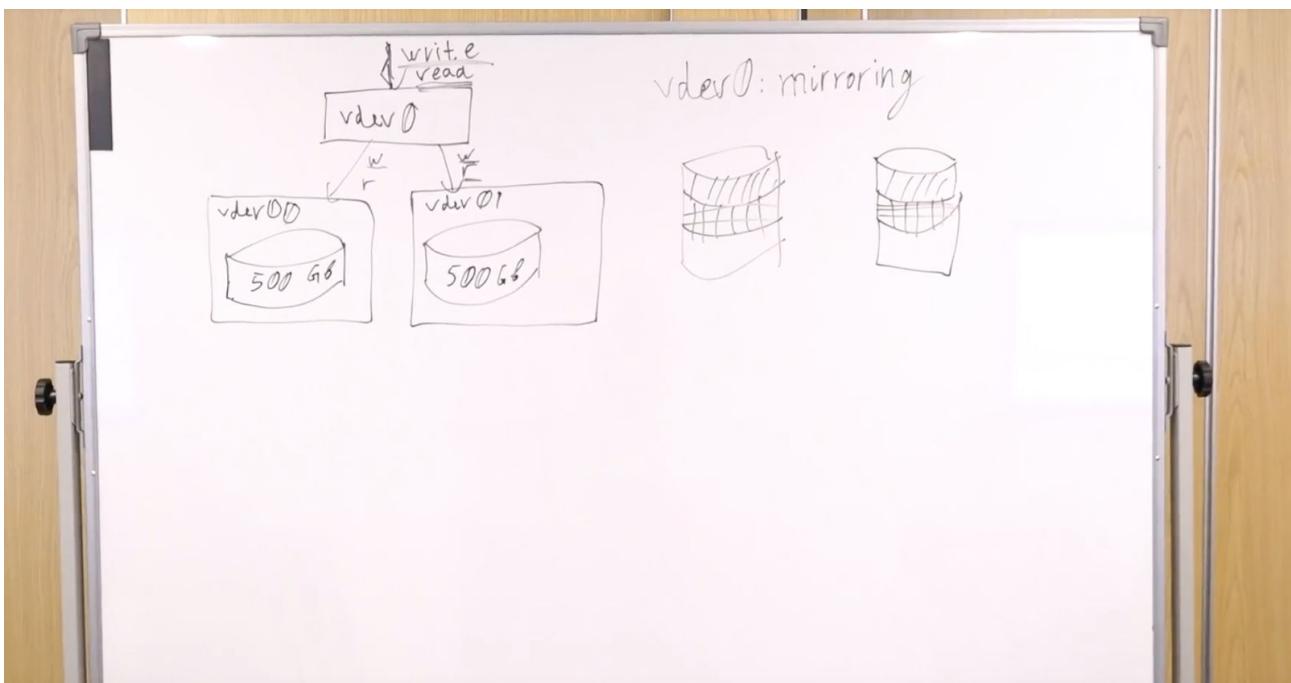
LVM (Logical Volume Manager)

Часто возникает необходимость распределять пространства жёстких дисков по логическим томам.
Одно из решений – это на этапе установки нового диска (или установки ОС на нём) разбить его на несколько разделов.
Проблема: размер раздела не всегда можно изменить
Проблема: при таком подходе невозможно создать том больше, чем размер одного жёсткого диска

Эту проблему решает LVM, он позволяет объединить несколько жёстких дисков в один Logical Volume Group, а его уже с помощью LVM можно разбить на логические тома, операции с которыми (например, расширение) производить гораздо проще



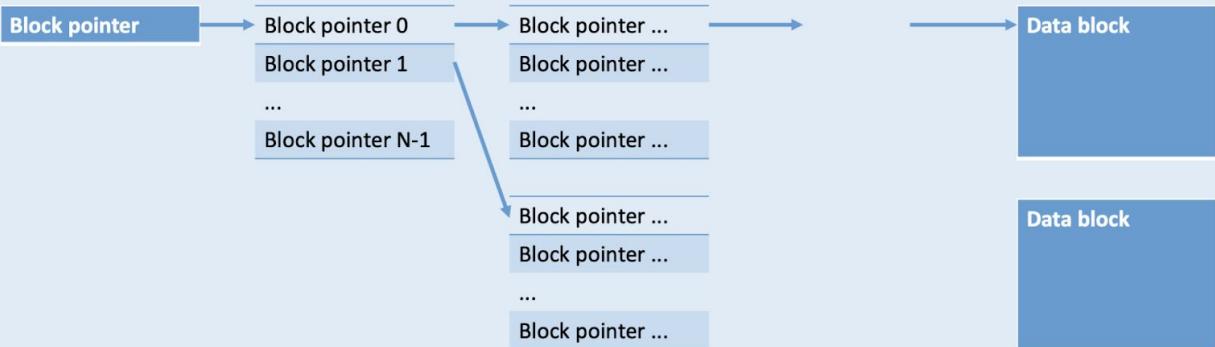
Acronis @ МФТИ



Основы построения файловых систем

Data management unit в ZFS: "объект" файловой системы

Один "объект файловой системы" или dnode, представляет собой дерево из указателей на блоки:



Замечание: в отличие от inode, содержащей direct block pointers, double indirect pointers и triple indirect pointers, dnode в ZFS имеет только один указатель на данные. Если блок данных имеет размер до 128Mb, то это будет прямой указатель на данные, иначе – указатель на дерево.

В структурах этого дерева лежат DMU

Основы построения файловых систем

Data management unit в ZFS

DMU предоставляет механизм хранения «файлов».

ZFS block pointer						
vdev0		grid	asize			
G	offset0					
G	vdev1		grid	asize		
G	offset1					
G	vdev2		grid	asize		
G	offset2					
BDX	lvl	type	cksum	comp	psize	lsize
spare						
spare						
physical birth time						
logical birth time						
fill count						
checksum[0]						
checksum[1]						
checksum[2]						
checksum[3]						

- vdev – идентификатор vdev, на котором располагается блок,
- grid – информация о типе raidz (не используется),
- asize – размер блока с учётом заголовков ФС,
- offset – смещение внутри vdev,
- G – флаг "gang block", блок, собранный SPA из нескольких меньших по размеру,
- B – флаг "big endian",
- D – флаг "deduped",
- X – не используемый флаг,
- lvl – уровень косвенности данного указателя,
- type – тип объекта DMU (block pointer, master node, file data, ZAP, quota info, filesystem metadata, etc),
- cksum – тип контрольной суммы,
- psize и lsize – физический и логический размеры блока,
- physical birth time – номер транзакции, создавшей блок,
- logical birth time – номер транзакции, создавшей блок, для дедуплицированных блоков,
- fill count – количество ненулевых блоков, на которые ссылается этот указатель

Основы построения файловых систем

Data management unit в ZFS: файлы, каталоги и файловые системы

Файл в ZFS – это обычновенный “объект” ФС.

Каталог – это “объект” ФС, который трактуется как хеш-таблица, отображающая имя файла в его dnode.

Замечание о терминологии: хеш-таблицы в ZFS обрабатывает модуль ZAP -- ZFS Attribute Processor, поэтому в документации можно встретить ссылки к “ZAP objects”.

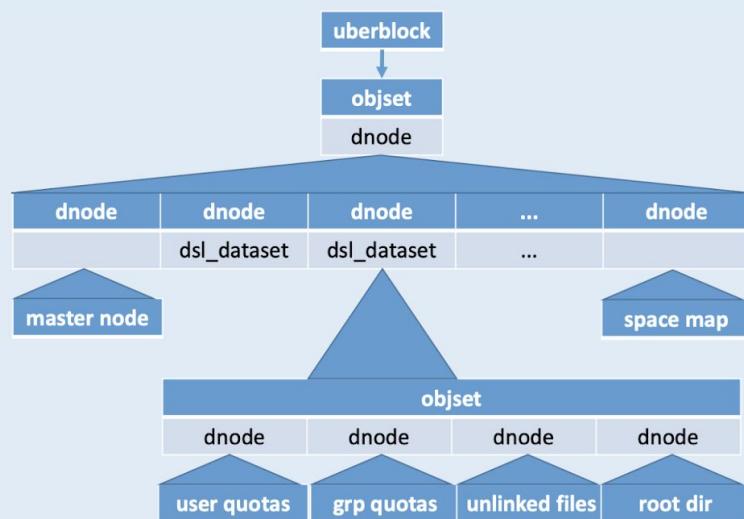
Файловая система в ZFS – это “объект ФС”, который состоит из четырёх ссылок:

- ZAP object корневого каталога,
- ZAP object, отслеживающий квоты пользователей,
- ZAP object, отслеживающий квоты групп,
- ZAP object, отслеживающий открытые файлы, не имеющие имени.

Следствие: создавать FS, разделяющие ресурсы одного пула дисков, в ZFS так же просто, как создать каталог или файл.

Основы построения файловых систем

Общий вид ZFS-пула



Основы построения файловых систем

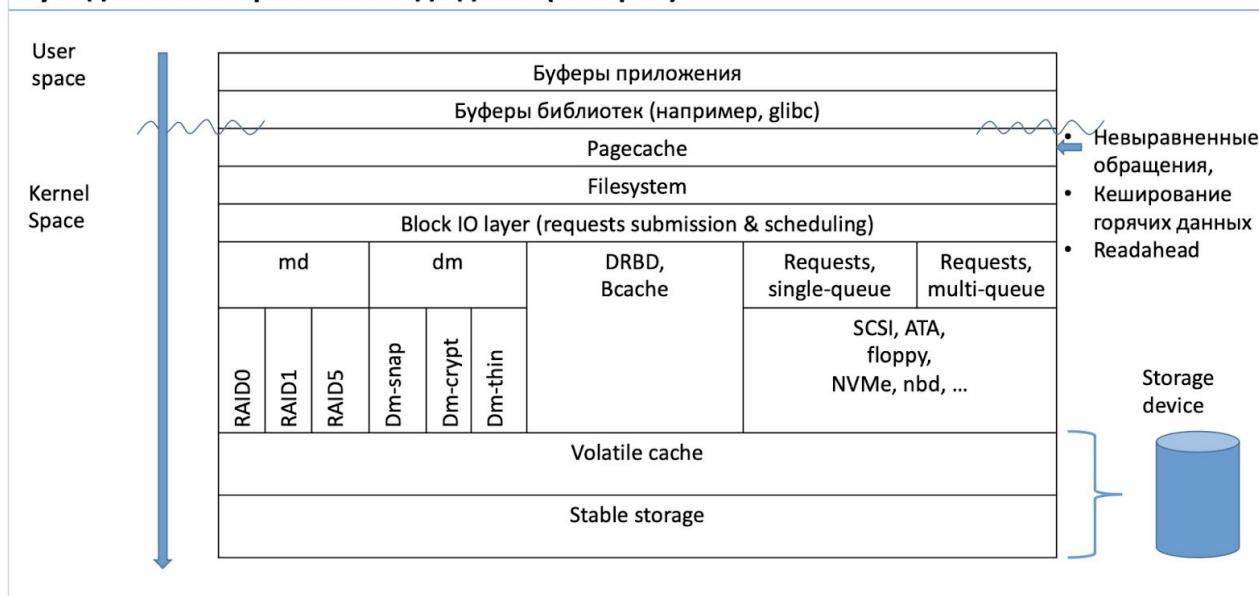
Требования к ФС и что нам даёт Copy-on-write

ФС всегда должна быть в согласованном состоянии.	Суперблок всегда указывает на целостную ФС.
Быстрый FSCK или, что лучше, отсутствие оного.	FSCK не нужен.
Быстрая запись в файлы и быстрая модификация метаданных.	Блоки-копии можно выписывать последовательно, притом неважно, принадлежат они одному файлу или разным.
Гибкое управление размером ФС, возможность использовать несколько дисков одновременно для большей надёжности или скорости.	SPA и ФС, которые являются файлами с точки зрения DMU.
Быстрые снимки состояния ФС, клоны ФС и откат к предыдущему состоянию.	Создание снимка бесплатно: надо просто не удалить старый суперблок, а сохранить ссылку на корень ФС как ссылку на корень снапшота.
Защита от случайных повреждений содержимого дисков.	Криптографические хеши в качестве контрольных сумм, организация всех данных пула в виде дерева Меркле.
Защита от RAID write hole.	RAID write hole возникают при перезаписи блоков, а в Copy-on-write FS перезаписи никогда не происходят.

24.

Основы построения файловых систем

Путь данных от приложения до диска (обзорно)



25.

Способ передавать файловые системы по сетям.

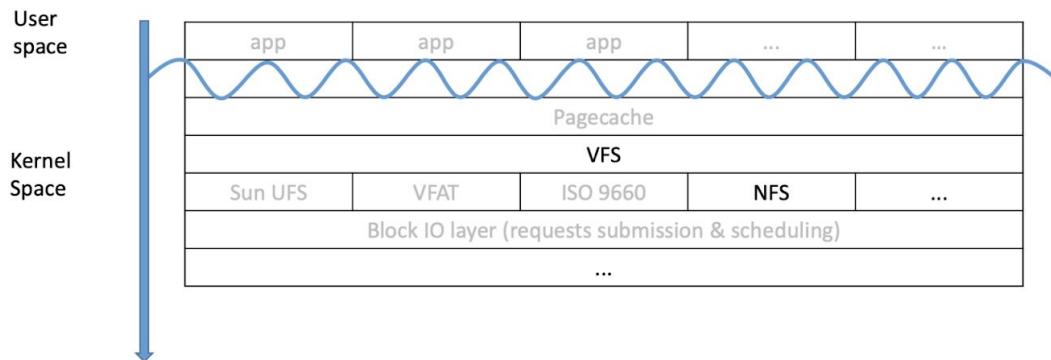
Клиентская машина исполняет симлинк на себе, а не на сервере.

Acronis @ МФТИ

Основы построения файловых систем

Основные цели

- Доступ к ФС по сети должен быть таким же, как доступ к локальной ФС, притом даже на уровне ядра:
NFS должен предоставлять те же колбеки для VFS, что и локальные файловые системы.



Короче тут нужна вся презентация.

```
ssize_t pread(int fd, void *buf, size_t count, off_t offset);  
ssize_t pwrite(int fd, const void *buf, size_t count, off_t offset);
```

ОПИСАНИЕ

pread() записывает максимум *count* байтов из описателя файлов *fd*, начиная со смещения *offset* (от начала файла), в буфер *buf*. Текущая позиция файла не изменяется.

pwrite() записывает максимум *count* байтов из буфера *buf* в описатель файла *fd*, начиная со смещения *offset*. Текущая позиция файла не изменяется.

Файл, заданный в *fd*, должен позволять изменение смещения.

ВОЗВРАЩАЕМЫЕ ЗНАЧЕНИЯ

При удачном завершении вызова возвращается количество прочитанных или записанных байтов (0 в случае выполнения функции **pwrite** означает, что никакой информации не было записано, а в случае выполнения функции **pread** - конец файла). При ошибке возвращается -1, а переменной *errno* присваивается номер ошибки.

`close` - закрыть файловый дескриптор

ОБЗОР

```
#include <unistd.h>
```

```
int close(int fd);
```

ОПИСАНИЕ

`close` закрывает файловый дескриптор, который после этого не ссылается ни на один и файл и может быть использован повторно. Все блокировки, находящиеся на соответствующем файле, снимаются (независимо от того, был ли использован для установки блокировки именно этот файловый дескриптор).

Если `fd` является последней копией какого-либо файлового дескриптора, то ресурсы, связанные с ним, освобождаются; если дескриптор был последней ссылкой на файл, удаленный с помощью `unlink(2)`, то файл окончательно удаляется.

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

`close` возвращает ноль при успешном завершении или -1, если произошла ошибка.

`stat`, `fstat`, `Istat` - считывает статус файла

СИНТАКСИС

```
#include <sys/types.h>
```

```
#include <sys/stat.h>
```

```
#include <unistd.h>
```

```
int stat(const char *file_name, struct stat *buf);
```

```
int fstat(int filedes, struct stat *buf);
```

```
int Istat(const char *file_name, struct stat *buf);
```

ОПИСАНИЕ

Эти функции возвращают информацию об указанном файле. Для этого не требуется иметь права доступа к файлу, хотя потребуются права поиска во всех каталогах, указанных в полном имени файла.

`stat` возвращает информацию о файле `file_name` и заполняет буфер `buf`. `Istat` идентична `stat`, но в случае символьных ссылок она возвращает информацию о самой ссылке, а не о файле, на который она указывает. `fstat` идентична `stat`, только

возвращается информация об открытом файле, на который указывает *filedes* (возвращаемый `open(2)`), а не о *file_name*.

ВОЗВРАЩАЕМЫЕ ЗНАЧЕНИЯ

В случае успеха возвращается ноль. При ошибке возвращается -1, а переменной *errno* присваивается номер ошибки.

`link` - make a new name for a file

SYNOPSIS

```
#include <unistd.h>
```

```
int link(const char *oldpath, const char *newpath);
```

DESCRIPTION

`link()` creates a new link (also known as a hard link) to an existing file.

If *newpath* exists it will *not* be overwritten.

This new name may be used exactly as the old one for any operation; both names refer to the same file (and so have the same permissions and ownership) and it is impossible to tell which name was the "original".

RETURN VALUE

On success, zero is returned. On error, -1 is returned, and *errno* is set appropriately.

`unlink` - удаляет имя и возможно файл, на который оно ссылается

ОБЗОР

```
#include <unistd.h>
```

```
int unlink(const char *pathname);
```

ОПИСАНИЕ

`unlink` удаляет имя из файловой системы. Если это имя было последней ссылкой на файл и больше нет процессов, которые держат этот файл открытым, данный файл удаляется и место, которое он занимает освобождается для дальнейшего использования.

Если имя было последней ссылкой на файл, но какие-либо процессы всё ещё держат этот файл открытым, файл будет оставлен пока последний файловый дескриптор, указывающий на него, не будет закрыт.

Если имя указывает на символьную ссылку, ссылка будет удалена.

Если имя указывает на сокет, FIFO или устройство, имя будет удалено, но процессы, которые открыли любой из этих объектов могут продолжать его использовать.

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

В случае успеха возвращается ноль. В случае ошибки возвращается -1 и значение *errno* устанавливается соответствующим образом.

`symlink` - создать новое имя для файла

ОБЗОР

```
#include <unistd.h>
```

```
int symlink(const char *topath, const char *frompath);
```

ОПИСАНИЕ

`symlink` создает символьную ссылку, которая называется *frompath* и содержит строку *topath*.

Символьные ссылки интерпретируются "на лету", как будто бы содержимое ссылки было подставлено вместо пути, по которому идет поиск файла или каталога.

Символьные ссылки могут содержать такие компоненты пути, как .. которые, (если используются в начале ссылки), ссылаются на родительский каталог того каталога, в котором находится ссылка.

Символьная ссылка (также известная как "мягкая ссылка") может указывать как на существующий, так и на несуществующий файлы; в последнем случае такая ссылка называется "висячей".

Права доступа к символьной ссылке не используются; её владелец игнорируется при поиске по ссылке, но проверяется при удалении или переименовании ссылки, находящейся в каталоге с установленным sticky битом.

Если *newpath* существует, он не будет перезаписан.

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

В случае успеха возвращается ноль. При ошибке возвращается -1, а значение *errno* устанавливается должным образом.

mmap, **munmap** - отражает файлы или устройства в память или снимает их отражение

СИНТАКСИС

```
#include <unistd.h>
#include <sys/mman.h>

#ifndef _POSIX_MAPPED_FILES

void * mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset);

int munmap(void *start, size_t length);

#endif
```

ОПИСАНИЕ

Функция **mmap** отражает *length* байтов, начиная со смещения *offset* файла (или другого объекта), определенного файловым описателем *fd*, в память, начиная с адреса *start*. Последний параметр (адрес) необязателен, и обычно бывает равен 0. Настоящее местоположение отраженных данных возвращается самой функцией **mmap**, и никогда не бывает равным 0.

ВОЗВРАЩАЕМЫЕ ЗНАЧЕНИЯ

При удачном выполнении **mmap** возвращает указатель на область с отраженными данными. При ошибке возвращается значение **MAP_FAILED** (-1), а переменная *errno* приобретает соответствующее значение. При удачном выполнении **munmap** возвращаемое значение равно нулю. При ошибке возвращается -1, а переменная *errno* приобретает соответствующее значение. (Вероятнее всего, это будет **EINVAL**).

Аргумент *prot* описывает желаемый режим защиты памяти (он не должен конфликтовать с режимом открытия файла). Оно является либо **PROT_NONE** либо побитовым ИЛИ одного или нескольких флагов **PROT_***.

PROT_EXEC

(данные в страницах могут исполняться);

PROT_READ

(данные можно читать);

PROT_WRITE

(в эту область можно записывать информацию);

PROT_NONE

(доступ к этой области памяти запрещен).

Параметр *flags* задает тип отражаемого объекта, опции отражения и указывает, принадлежат ли отраженные данные только этому процессу или их могут читать другие. Он состоит из комбинации следующих битов:

MAP_FIXED

Не использовать другой адрес, если адрес задан в параметрах функции. Если заданный адрес не может быть использован, то функция `mmap` вернет сообщение об ошибке. Если используется `MAP_FIXED`, то `start` должен быть пропорционален размеру страницы. Использование этой опции не рекомендуется.

MAP_SHARED

Разделить использование этого отражения с другими процессами, отражающими тот же объект. Запись информации в эту область памяти будет эквивалентна записи в файл. Файл может не обновляться до вызова функций `msync(2)` или `munmap(2)`.

MAP_PRIVATE

Создать неразделяемое отражение с механизмом copy-on-write. Запись в эту область памяти не влияет на файл. Не определено, являются или нет изменения в файле после вызовов `mmap` видимыми в отраженном диапазоне.