

# Основы построения файловых систем



# Log-structured File System\*

\* M. Rosenblum, J. Ousterhout: *The design and implementation of a log-structured file system*,  
\* <https://www.usenix.org/system/files/conference/fast15/fast15-paper-lee.pdf>

## Создание файла и количество требуемых для этого операций

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.

## Создание файла и количество требуемых для этого операций

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:

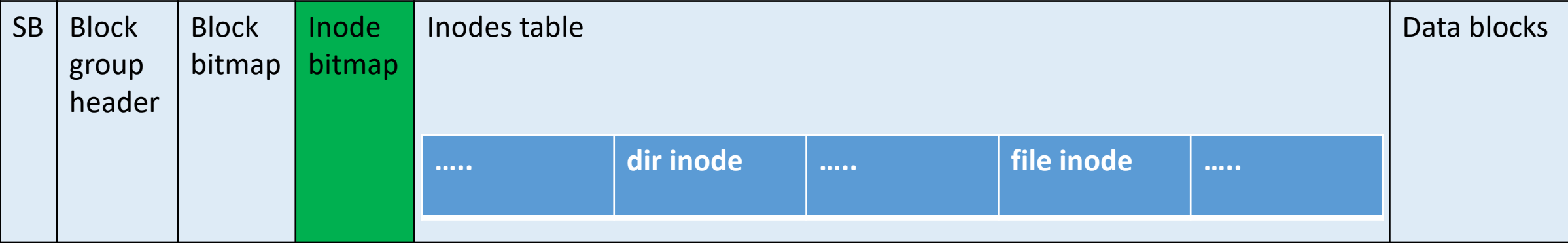
SB	Block group header	Block bitmap	Inode bitmap	Inodes table					Data blocks
				.....	dir inode	.....	file inode	.....	

# Создание файла и количество требуемых для этого операций

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:



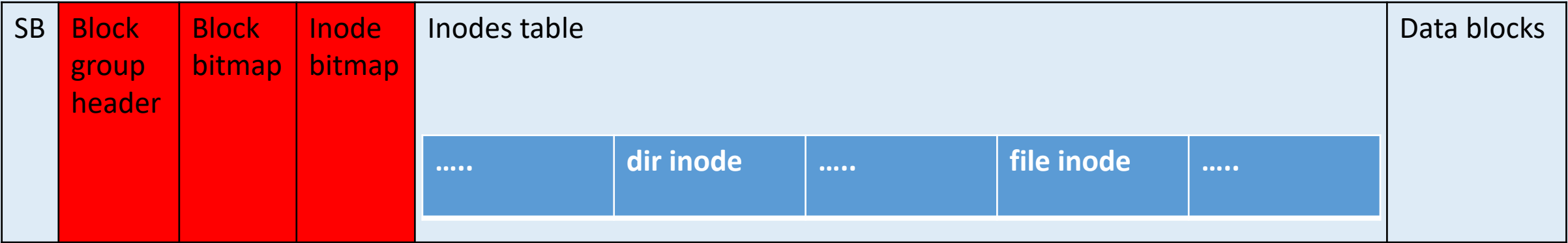
## Создание файла и количество требуемых для этого операций

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:



## Создание файла и количество требуемых для этого операций

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:

SB	Block group header	Block bitmap	Inode bitmap	Inodes table	Data blocks
				<div><div>.....</div><div>dir inode</div><div>.....</div><div>file inode</div><div>.....</div></div>	


## Создание файла и количество требуемых для этого операций

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:

SB	Block group header	Block bitmap	Inode bitmap	Inodes table					Data blocks
				.....	dir inode	.....	file inode	.....	



## Создание файла и количество требуемых для этого операций

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:


SB	Block group header	Block bitmap	Inode bitmap	Inodes table	Data blocks
				<div><div>.....</div><div>dir inode</div><div>.....</div><div>file inode</div><div>.....</div></div>	

## Создание файла и количество требуемых для этого операций

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.
- 

Итого, надо сделать записи в **четыре** области диска, не идущие подряд, т.е. нужны 4 перемещения головки диска.

# Создание файла и количество требуемых для этого операций

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.

Итого, надо сделать записи в четыре области диска, не идущие подряд, т.е. нужны 4 перемещения головки диска.

У файловых систем с журналированием получается лучше: транзакцию можно подтвердить как успешную сразу после записи в журнал. Т.е., задержка на вызов `open()` составляет одно перемещение головки диска. Но затем в фоне всё равно надо унести данные из журнала.

## Создание файла и количество требуемых для этого операций

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.

Итого, надо сделать записи в четыре области диска, не идущие подряд, т.е. нужны 4 перемещения головки диска.

У файловых систем с журналированием получается лучше: транзакцию можно подтвердить как успешную сразу после записи в журнал. Т.е., задержка на вызов `open()` составляет одно перемещение головки диска. Но затем в фоне всё равно надо унести данные из журнала.

**Идея:** зачем уносить данные из журнала в обычные области ФС? Можно ли всё доступное место трактовать как журнал?

## Log-structured FS

Организация ФС в виде лога ставит несколько вопросов:

1. Как находить иноды в такой ФС? В частности, где найти файл, содержащий список элементов в корневом каталоге?
2. Запись в журнал ext3 происходит быстро потому, что она линейная. Журнал в LFS тоже должен обеспечивать линейные записи, он не должен фрагментироваться.

## Log-structured FS

Организация ФС в виде лога ставит несколько вопросов:

1. Как находить иноды в такой ФС? В частности, где найти файл, содержащий список элементов в корневом каталоге?
  - Можно сканировать журнал в поиске операций добавления/удаления/переименования файлов. Это плохо, поскольку просканировать историю ФС от момента её создания займёт много времени.
  - В журнал можно писать inode maps, таблицы со ссылками на области дисков, содержащие иноды. А указатели на inode maps хранить в некоторой фиксированной области диска.
2. Запись в журнал ext3 происходит быстро потому, что она линейная. Журнал в LFS тоже должен обеспечивать линейные записи, он не должен фрагментироваться.

## Log-structured FS

Организация ФС в виде лога ставит несколько вопросов:

1. Как находить иноды в такой ФС? В частности, где найти файл, содержащий список элементов в корневом каталоге?
  - Можно сканировать журнал в поиске операций добавления/удаления/переименования файлов. Это плохо, поскольку просканировать историю ФС от момента её создания займёт много времени.
  - В журнал можно писать inode maps, таблицы со ссылками на области дисков, содержащие иноды. А указатели на inode maps хранить в некоторой фиксированной области диска.
2. Запись в журнал ext3 происходит быстро потому, что она линейная. Журнал в LFS тоже должен обеспечивать линейные записи, он не должен фрагментироваться.

# Log-structured FS

Организация ФС в виде лога ставит несколько вопросов:

1. Как находить иноды в такой ФС? В частности, где найти файл, содержащий список элементов в корневом каталоге?
2. Запись в журнал ext3 происходит быстро потому, что она линейная. Журнал в LFS тоже должен обеспечивать линейные записи, он не должен фрагментироваться.

Рассмотрим такой пример:

Для простоты будем считать, что заголовок имеет длину 1Kb.



1. Создадим файлы file0, file1 и file2 длиной 1Kb каждый.
2. Удалим file1.
3. Создадим файл file3 длиной 3Kb.



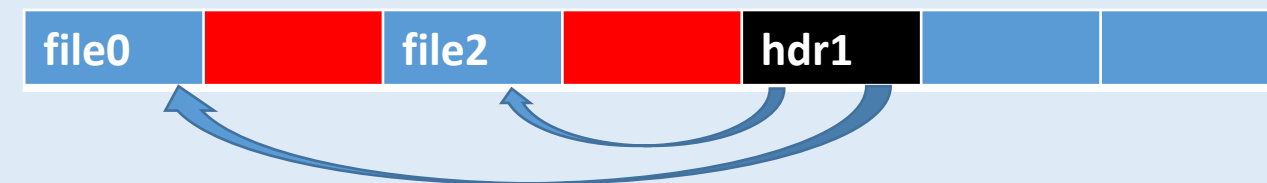
# Log-structured FS

Организация ФС в виде лога ставит несколько вопросов:

1. Как находить иноды в такой ФС? В частности, где найти файл, содержащий список элементов в корневом каталоге?
2. Запись в журнал ext3 происходит быстро потому, что она линейная. Журнал в LFS тоже должен обеспечивать линейные записи, он не должен фрагментироваться.

Рассмотрим такой пример:

Для простоты будем считать, что заголовок имеет длину 1Kb.



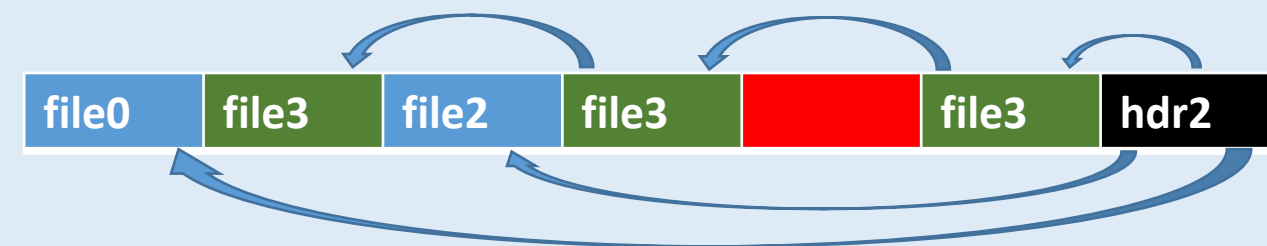
1. Создадим файлы file0, file1 и file2 длиной 1Kb каждый.
2. Удалим file1.
3. Создадим файл file3 длиной 3Kb.

# Log-structured FS

Организация ФС в виде лога ставит несколько вопросов:

1. Как находить иноды в такой ФС? В частности, где найти файл, содержащий список элементов в корневом каталоге?
2. Запись в журнал ext3 происходит быстро потому, что она линейная. Журнал в LFS тоже должен обеспечивать линейные записи, он не должен фрагментироваться.

Рассмотрим такой пример:



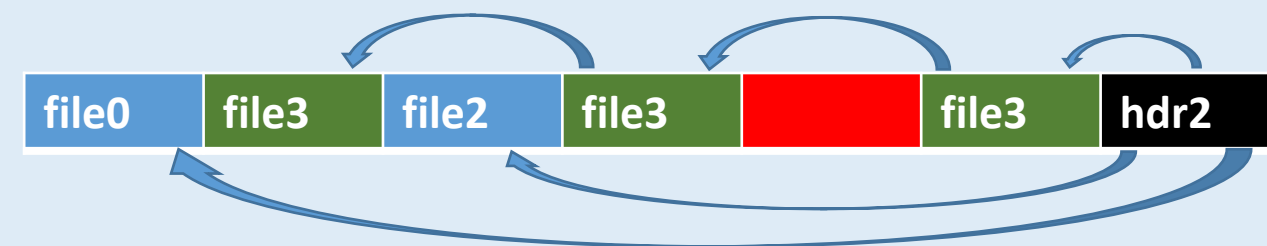
1. Создадим файлы file0, file1 и file2 длиной 1Kb каждый.
2. Удалим file1.
3. Создадим файл file3 длиной 3Kb.

# Log-structured FS

Организация ФС в виде лога ставит несколько вопросов:

1. Как находить иноды в такой ФС? В частности, где найти файл, содержащий список элементов в корневом каталоге?
2. Запись в журнал ext3 происходит быстро потому, что она линейная. Журнал в LFS тоже должен обеспечивать линейные записи, он не должен фрагментироваться.

Рассмотрим такой пример:



1. Создадим файлы file0, file1 и file2 длиной 1Kb каждый.
2. Удалим file1.
3. Создадим файл file3 длиной 3Kb.

**Наблюдение:** создание file3 привело к случайному IO: пришлось записать в три области на диске, не идущие подряд.

**Вопрос:** почему блок, содержащий hdr1, был помечен как пустой, но не переиспользован для записи file3?

## Log-structured FS

Организация ФС в виде лога ставит несколько вопросов:

1. Как находить иноды в такой ФС? В частности, где найти файл, содержащий список элементов в корневом каталоге?
  - В журнал можно писать inode maps, таблицы со ссылками на области дисков, содержащие иноды. А указатели на inode maps хранить в некоторой фиксированной области диска.
2. Запись в журнал ext3 происходит быстро потому, что она линейная. Журнал в LFS тоже должен обеспечивать линейные записи, он не должен фрагментироваться.
  - Длины записей в журнале должны быть ограничены снизу. Например, диск можно разбить на сегменты, каждый из которых можно перезаписывать только целиком\*.

*\* Обратите внимание на схожесть организации диска в LFS и того, как флеш-память разбивается на erase blocks.*

## Log-structured FS

Организация ФС в виде лога ставит несколько вопросов:

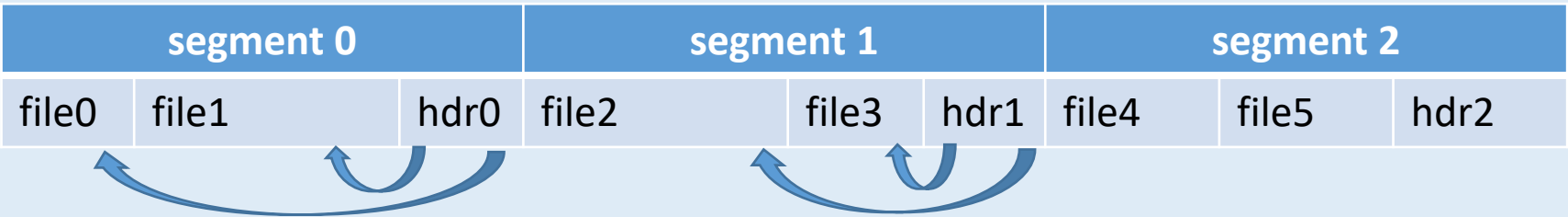
1. Как находить иноды в такой ФС? В частности, где найти файл, содержащий список элементов в корневом каталоге?
  - В журнал можно писать inode maps, таблицы со ссылками на области дисков, содержащие иноды. А указатели на inode maps хранить в некоторой фиксированной области диска.
2. Запись в журнал ext3 происходит быстро потому, что она линейная. Журнал в LFS тоже должен обеспечивать линейные записи, он не должен фрагментироваться.
  - Длины записей в журнале должны быть ограничены снизу. Например, диск можно разбить на сегменты, каждый из которых можно перезаписывать только целиком.
3. Как обрабатывать удаление файлов, которые короче одного сегмента журнала?

# Log-structured FS

Организация ФС в виде лога ставит несколько вопросов:

- 1. Как находить иноды в такой ФС? В частности, где найти файл, содержащий список элементов в корневом каталоге?
- 2. Запись в журнал ext3 происходит быстро потому, что она линейная. Журнал в LFS тоже должен обеспечивать линейные записи, он не должен фрагментироваться.
- 3. Как обрабатывать удаление файлов, которые короче одного сегмента журнала?

Рассмотрим такой пример:



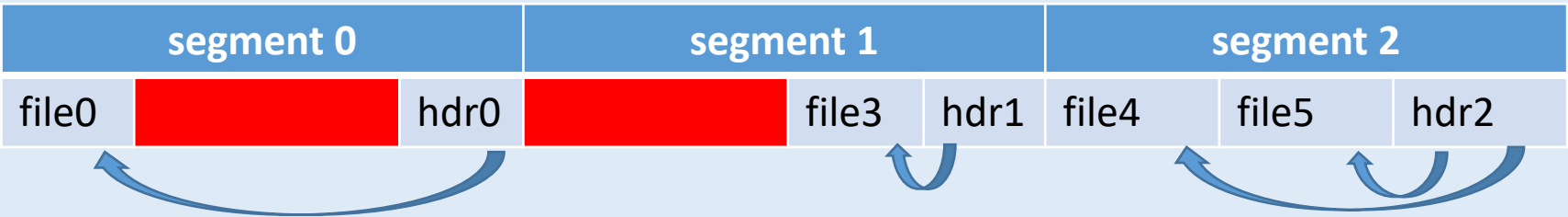
- 1. Создадим file0 и file1,
- 2. Создадим file2 и file3,
- 3. Удалим file1 и file2,
- 4. Создадим file4 и file5.

# Log-structured FS

Организация ФС в виде лога ставит несколько вопросов:

- 1. Как находить иноды в такой ФС? В частности, где найти файл, содержащий список элементов в корневом каталоге?
- 2. Запись в журнал ext3 происходит быстро потому, что она линейная. Журнал в LFS тоже должен обеспечивать линейные записи, он не должен фрагментироваться.
- 3. Как обрабатывать удаление файлов, которые короче одного сегмента журнала?

Рассмотрим такой пример:



- 1. Создадим file0 и file1,
- 2. Создадим file2 и file3,
- 3. Удалим file1 и file2,
- 4. Создадим file4 и file5.

**Проблема:** файлы file4 и file5 уместились бы в областях, которые занимали file1 и file2, но мы не имеем права перезаписать сегменты 0 и 1, поскольку там всё ещё есть данные других файлов.

## Log-structured FS

Организация ФС в виде лога ставит несколько вопросов:

1. Как находить иноды в такой ФС? В частности, где найти файл, содержащий список элементов в корневом каталоге?
  - В журнал можно писать inode maps, таблицы со ссылками на области дисков, содержащие иноды. А указатели на inode maps хранить в некоторой фиксированной области диска.
2. Запись в журнал ext3 происходит быстро потому, что она линейная. Журнал в LFS тоже должен обеспечивать линейные записи, он не должен фрагментироваться.
  - Длины записей в журнале должны быть ограничены снизу. Например, диск можно разбить на сегменты, каждый из которых можно перезаписывать только целиком.
3. Как обрабатывать удаление файлов, которые короче одного сегмента журнала?
  - Для сегментов журнала требуется сборка мусора.



## Слежение за свободным местом в LFS

В отличие от ext2, в LFS нет block bitmap или списка свободных/занятых блоков.

Для слежения за свободным местом используются segment summary block.

Segment summary block содержит список пар (inode, extent), которые хранятся в данном сегменте. По номеру иноды LFS может получить список всех экстентов в файле и проверить, входит ли в него экстент, сохранённый в данном сегменте.

Простая оптимизация на случай удаления файла: хранить тройки (inode, generation, extent), где generation – это целое число, которое увеличивается каждый раз, когда файл, соответствующий иноде, удаляется или обрезается до нулевого размера. Все блоки, у которых поколение меньше поколения иноды, гарантированно являются мусором.

## Checkpointing и восстановление после сбоев в LFS

После записи сегмента журнала LFS обновляет checkpoint region, в котором хранится указатель на последнюю транзакцию в журнале.

Checkpoint region имеется две штуки и они перезаписываются по порядку. Помимо указателя на голову журнала, регион содержит монотонно растущий номер транзакции. При монтировании ФС выбирается тот checkpoint region, у которого номер транзакции выше.

**Вопрос:** зачем надо два checkpoint region?

## Wandering tree problem

Как реализовать `fsync()` на отдельный файл в LFS? – Записать целую транзакцию в журнал.

**Проблема:** приложения вроде SQLite часто вызывают `fsync()` после записи небольшой порции данных.

Что будет происходить с журналом LFS в такой ситуации? – В нём будут появляться много коротких транзакций и размер метаданных ФС станет сопоставим (или больше) с размером пользовательских данных.

Источники накладных расходов: помимо пользовательских данных и иногда в журнал надо записать содержимое

- Каталога, содержащего изменившийся файл, и его иноду,
- и так до корня по всему пути к файлу,
- segment summary block,
- заголовок транзакции.

В F2FS при `fsync()` на отдельный файл в журнал выписываются только блоки данных файла и его инода, а длина этой записи сохраняется в checkpoint region (не путать с указателем на голову транзакции!). Позже, при записи транзакции она форматируется так, чтобы выписанные при `fsync()` блоки данных составляли часть транзакции.

Такое решение требует **roll forward** при монтировании файловой системы: после нахождения последней успешно применённой транзакции ФС должна

- применить изменения в страницах данных из неполной транзакции,
- дописать транзакцию до полной.

## Стратегии сбора мусора

1. Как выбирать сегменты, в которых собрать мусор?
  - убирать мусор из тех сегментов, где его больше всего,
  - убирать мусор из наиболее фрагментированных сегментов.
2. Как перегруппировывать данные из сегментов?
  - хватит ли просто сливать несколько фрагментированных сегментов в один?

## Стратегии сбора мусора

1. Как выбирать сегменты, в которых собрать мусор?
  - убирать мусор из тех сегментов, где его больше всего,
  - убирать мусор из наиболее фрагментированных сегментов.
2. Как перегруппировывать данные из сегментов?
  - хватит ли просто сливать несколько фрагментированных сегментов в один?

Наблюдение:

- Если просто объединять сегменты, в которых оказалось много незанятого места (например,  $\geq 50\%$ ), в один, то мы получаем большой write amplification: одни и те же данные приходится копировать во многих циклах сборки мусора.

## Стратегии сбора мусора

1. Как выбирать сегменты, в которых собрать мусор?
  - убирать мусор из тех сегментов, где его больше всего,
  - убирать мусор из наиболее фрагментированных сегментов.
2. Как перегруппировывать данные из сегментов?
  - хватит ли просто сливать несколько фрагментированных сегментов в один?

Наблюдение:

- Если просто объединять сегменты, в которых оказалось много незанятого места (например,  $\geq 50\%$ ), в один, то мы получаем большой write amplification: одни и те же данные приходится копировать во многих циклах сборки мусора.

Объяснение:

1. Файлы, которые редко изменяются (т.н. холодные данные), имеют тенденцию накапливаться в сегментах, которые заполнены лишь немного больше, чем до порога уборки мусора.
2. Таких сегментов накапливается много и GC приходится их обрабатывать не тогда, когда в них будет достаточно мусора, а тогда, когда на всей ФС будет оставаться мало места.
3. Новые сегменты с холодными данными снова будут подолгу подходить к порогу сборки мусора и их придётся очищать из-за исчерпания места на ФС, и т.д.

## Стратегии сбора мусора

1. Как выбирать сегменты, в которых собрать мусор?
  - убирать мусор из тех сегментов, где его больше всего,
  - убирать мусор из наиболее фрагментированных сегментов.
2. Как перегруппировывать данные из сегментов?
  - хватит ли просто сливать несколько фрагментированных сегментов в один?

Идея (generational garbage collection):

- для сегментов следует отслеживать, как давно были изменены данные в них,
- при сборке мусора надо группировать в выходные сегменты данные близких времён модификации.

Таким образом мы автоматически получаем разделение сегментов на

- горячие: данные часто перезаписываются, сегмент с большой вероятностью оказывается мусором целиком задолго до того, как система решит, что нужна сборка мусора,
- холодные: данные редко (или даже никогда) не перезаписываются и лежат в сегментах, которые заняты почти полностью; такие сегменты редко проходят через GC.

Холодные данные иногда можно определить сразу: например, таковыми являются все файлы с мультимедиа.