# Основы построения файловых систем


# Создание файла и исчезновение питания

# 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);
```

# 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.

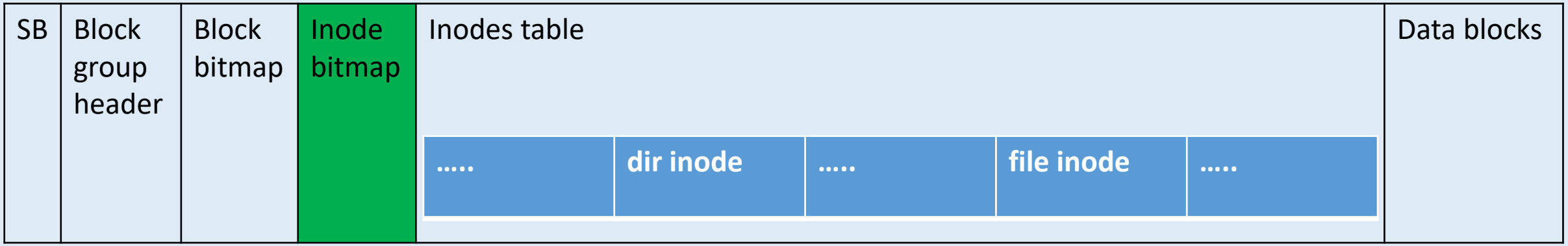
# Создание файла и исчезновение питания

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:



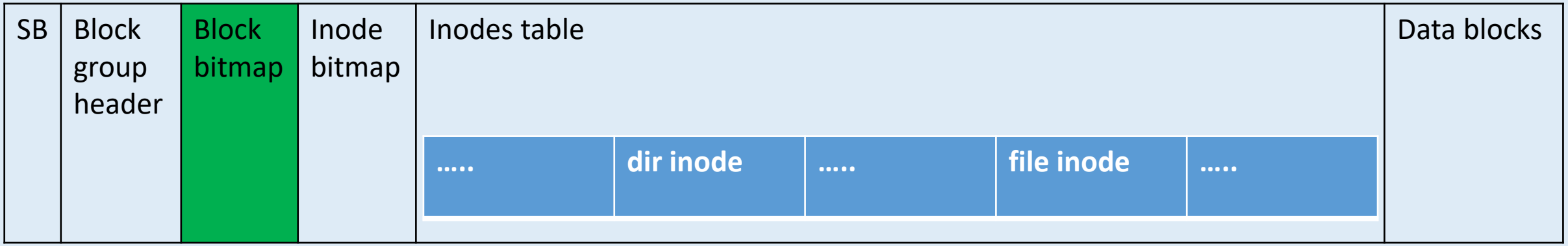
# Создание файла и исчезновение питания

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:



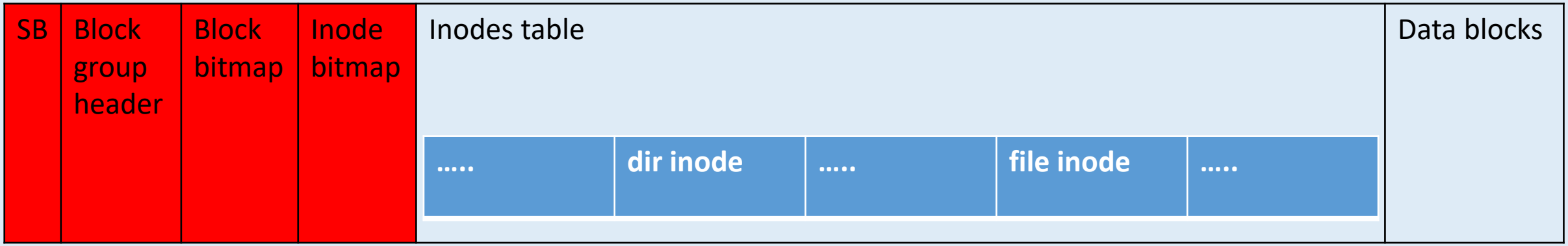
# Создание файла и исчезновение питания

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:



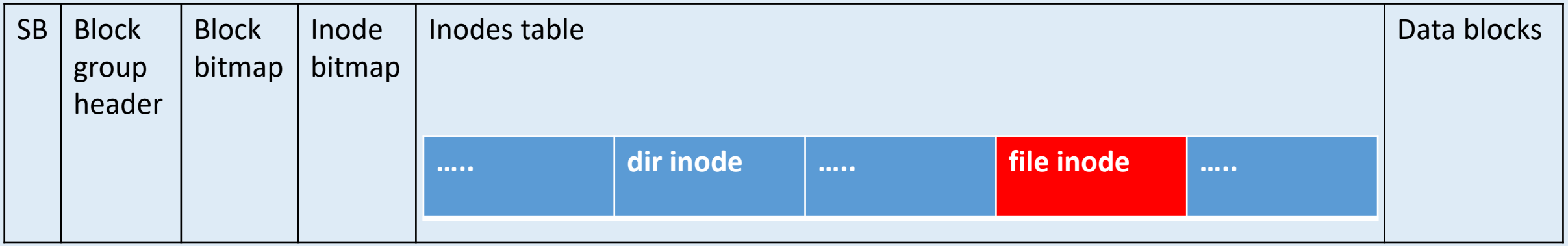
# Создание файла и исчезновение питания

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:



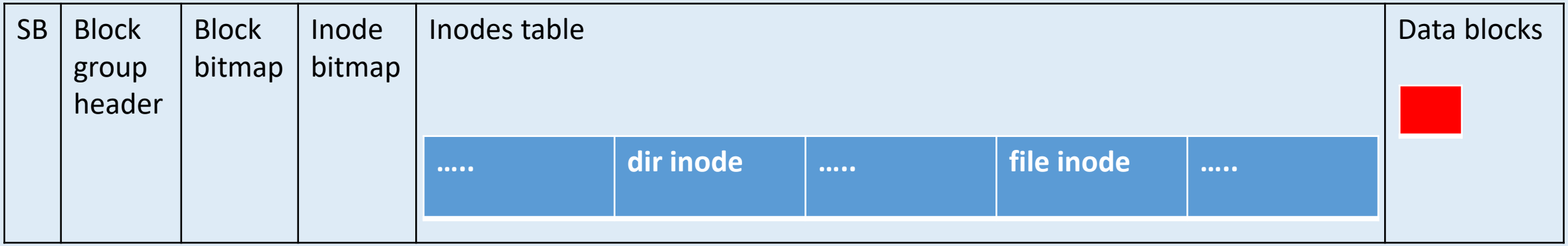
# Создание файла и исчезновение питания

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:



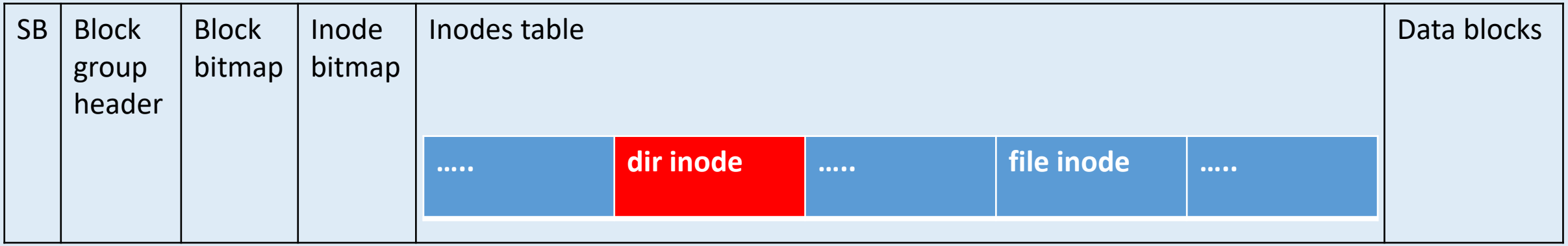
# Создание файла и исчезновение питания

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:



# 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.

# 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.

## Что должен обеспечить журнал на ФС

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

# Что должен обеспечить журнал на ФС

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

Упорядочивание:

Если приложение делает изменения в файле Y после fdatasync на файл X или изменения метаданных X, то после падения изменения в Y должны быть видны только если были применены изменения в X.

Пример: PostgreSQL, git и многие другие хранят данные в нескольких файлах, в которых есть ссылки друг на друга. Файл, на который указывает ссылка, создаётся до ссылки. После падения это свойство должно сохраниться.

# 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.

# 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.

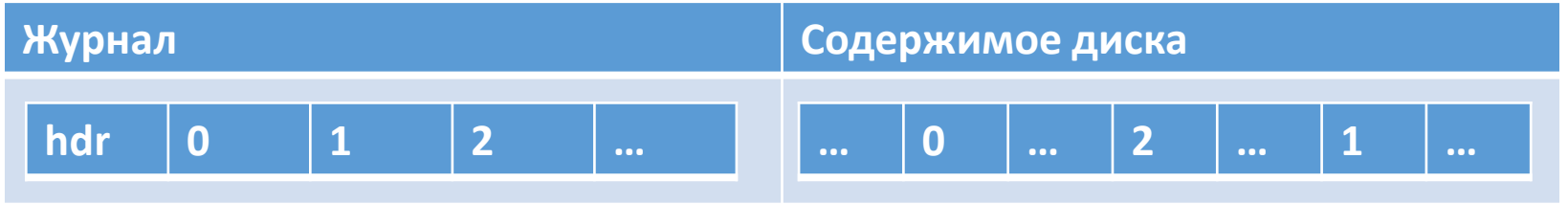
Теоретически, упорядочивания можно добиться, если писать блоки в нужном порядке, и каждый раз делать fsync() после записи. Но так будет слишком медленно.

# 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.

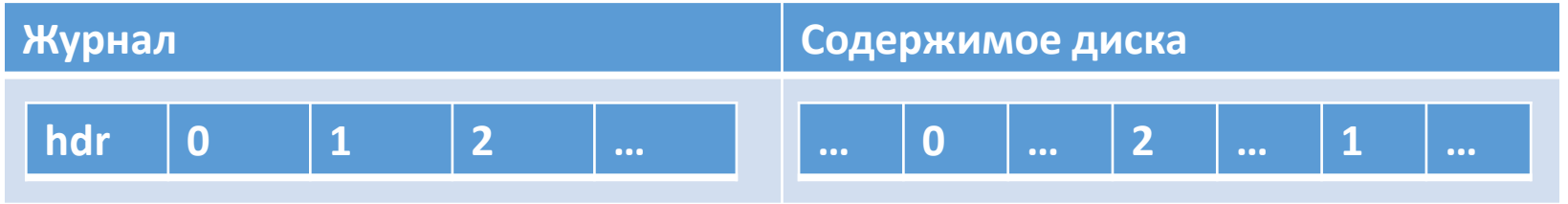


# Что должен обеспечить журнал на ФС

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.




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**.

# Что журналировать? Consistent FS state.

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

# 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.

# 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.

# 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 соответсвующие 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?

# Что журналировать? Consistent FS state.

Рассмотрим пример добавления данных в конец файла на XFS.

XFS

- Найдёт свободные нулевые блоки,
- Зажурналирует 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?

# Что журналировать? Consistent FS state.

Рассмотрим пример добавления данных в конец файла на XFS.

XFS

- Найдёт свободные нулевые блоки,
- Зажурналирует 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.

# Что журналировать? Consistent FS state.

Рассмотрим пример добавления данных в конец файла на XFS.

XFS

- Найдёт свободные нулевые блоки,
- Зажурналирует 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 одна причина для пользовательского приложения делать fsync() - ошибки отложенного writeback.

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

# Checkpointing

Журнал ФС имеет ограниченную длину, его надо чистить.

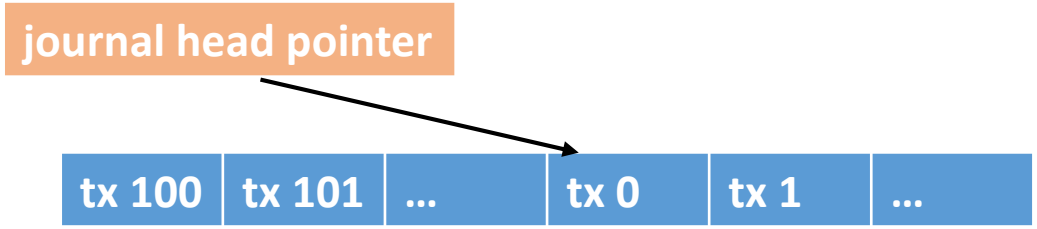
Для этого служит процедура checkpointing:

- Записать на диск группу транзакций,
- fsync(),
- Продвинуть указатель на голову журнала (reclaim journal space).

Запись может проводиться по таймеру или при накоплении достаточного числа транзакций в журнале.

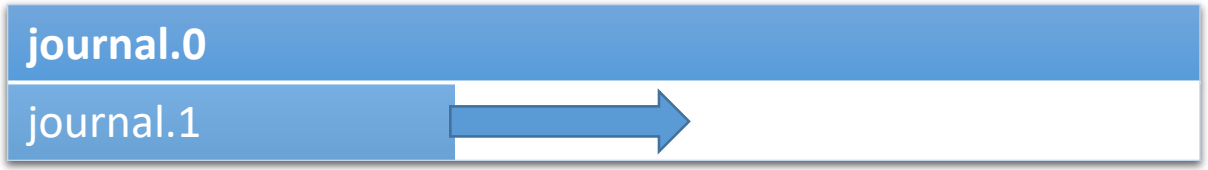
# Два способа хранения журнала

Ring buffer




При достижении конца журнала начинаем писать с начала.
Важно следить за тем, чтобы не перетереть транзакции, которые ещё не закоммичены.

В нескольких файлах
(не для журнала ФС, а для журналов приложений)

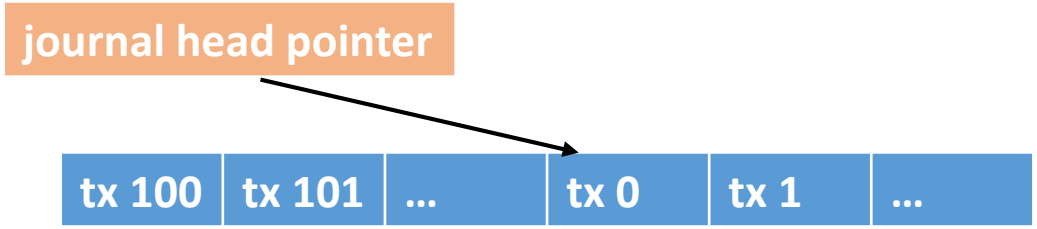



При переполнении одного файла начинаем писать следующий по номеру.

По мере коммита транзакций удаляем старые журналы.

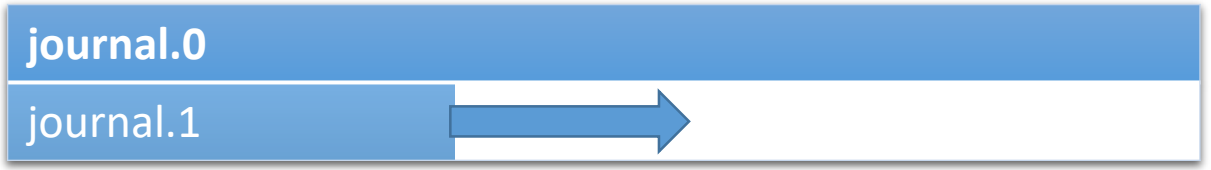
# Два способа хранения журнала

Ring buffer




При достижении конца журнала начинаем писать с начала. Важно следить за тем, чтобы не перетереть транзакции, которые ещё не закоммичены.

В нескольких файлах
(не для журнала ФС, а для журналов приложений)




При переполнении одного файла начинаем писать следующий по номеру.

По мере коммита транзакций удаляем старые журналы.

Примечание: полезно не просто дописывать в конец файла, а резервировать место в файле журнала, чтобы использовать fdatasync() вместо fsync() *.

** Почему fallocate() + fdatasync() может быть в несколько раз быстрее fsync()?*

# 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 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.

Журналирование логических изменений имеет плюс: размер inode в ext4 равен 256b, а если журналировать изменения **блоков**, то минимальная запись в журнале займёт 4k (типичный размер блока).

# 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?

# 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”)?

# 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.

# 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.

# Защита от неатомарности записи в журнал

Проблема: запись на диск более, чем одного сектора, не является атомарной операцией. Как гарантировать, что в середине транзакции не будет мусора?

# Защита от неатомарности записи в журнал

Проблема: запись на диск более, чем одного сектора, не является атомарной операцией. Как гарантировать, что в середине транзакции не будет мусора?

ext4

Каждая запись в журнале имеет следующий формат:

| header:<br>• magic,<br>• len*,<br>• csum | tx content | footer |
|---|---|---|

После выписывания содержимого транзакции в журнал делаем fsync, затем пишем footer.

Если у транзакции нет footer-блока или не сошлась контрольная сумма, то считаем, что эту транзакцию мы полностью не выписали на диск, т.е. мы дошли до конца журнала.

** В ext4 этого поля в журнале нет; надеются на magic number из footer block.*

# Защита от неатомарности записи в журнал

Проблема: запись на диск более, чем одного сектора, не является атомарной операцией. Как гарантировать, что в середине транзакции не будет мусора?

ext4

Каждая запись в журнале имеет следующий формат:

| header:<br>• magic,<br>• len*,<br>• csum | tx content | footer |
|---|---|---|

После выписывания содержимого транзакции в журнал делаем fsync, затем пишем footer.

Если у транзакции нет footer-блока или не сошлась контрольная сумма, то считаем, что эту транзакцию мы полностью не выписали на диск, т.е. мы дошли до конца журнала.

XFS

В журнал пишутся секторы, которые надо модифицировать. Сектор начинается со счётчика числа монтирований ФС:

| 8; sector1 | 8; sector2 | … | 7; sector0 |
|---|---|---|---|

Запись сектора атомарна, поэтому место обрыва журнала однозначно определяется однозначно как место немонотонности счётчика.

Транзакция имеет вид:

- сектор с заголовком и головами остальных секторов,
- секторы, изменённые в транзакции.

** В ext4 этого поля в журнале нет; надеются на magic number из footer block.*