The basics of file systems

Neapolis University Pafos

JET BRAINS

Imagine your machine crash when creating a file

Imagine your machine crash when creating a file

Creating a file is a simple operation from the point of view of a user:

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

Imagine your machine crash when creating a file

Creating a file is a simple operation from the point of view of a user:

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

A file system, on the other hand, performs multiple steps:

1. Find a free inode,
2. Find free blocks for the file content,
3. Flag the inode and the data blocks as in-use,
4. Fill the inode,
5. Append a struct ext2_dir_entry to the parent directory of the new file,
6. Since the directory length has grown, the FS has to update the inode of the directory.

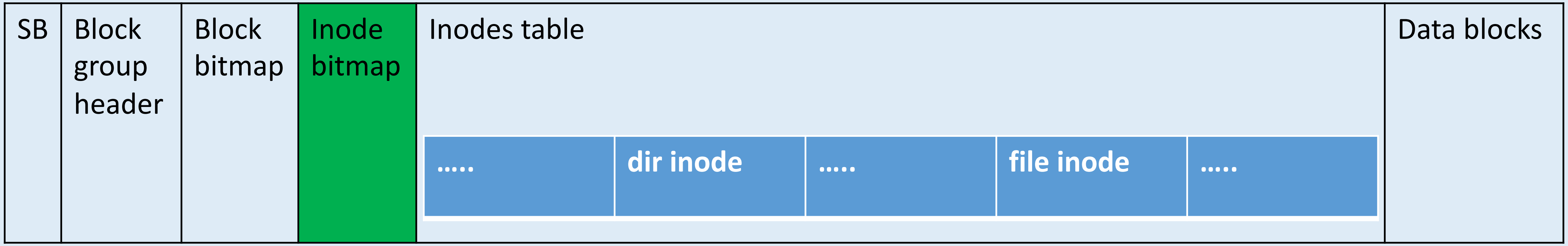
Imagine your machine crash when creating a file

Creating a file is a simple operation from the point of view of a user:

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

A file system, on the other hand, performs multiple steps:

1. Find a free inode,
2. Find free blocks for the file content,
3. Flag the inode and the data blocks as in-use,
4. Fill the inode,
5. Append a struct ext2_dir_entry to the parent directory of the new file,
6. Since the directory length has grown, the FS has to update the inode of the directory.



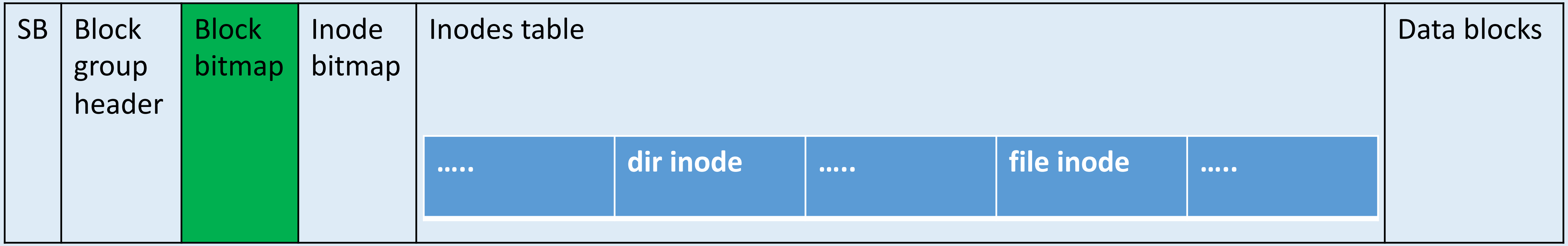
Imagine your machine crash when creating a file

Creating a file is a simple operation from the point of view of a user:

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

A file system, on the other hand, performs multiple steps:

1. Find a free inode,
2. Find free blocks for the file content,
3. Flag the inode and the data blocks as in-use,
4. Fill the inode,
5. Append a struct ext2_dir_entry to the parent directory of the new file,
6. Since the directory length has grown, the FS has to update the inode of the directory.



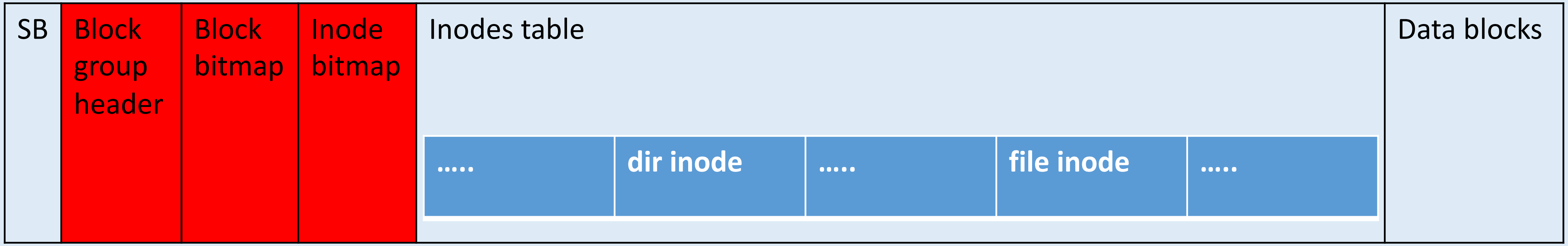
Imagine your machine crash when creating a file

Creating a file is a simple operation from the point of view of a user:

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

A file system, on the other hand, performs multiple steps:

1. Find a free inode,
2. Find free blocks for the file content,
3. Flag the inode and the data blocks as in-use,
4. Fill the inode,
5. Append a struct ext2_dir_entry to the parent directory of the new file,
6. Since the directory length has grown, the FS has to update the inode of the directory.



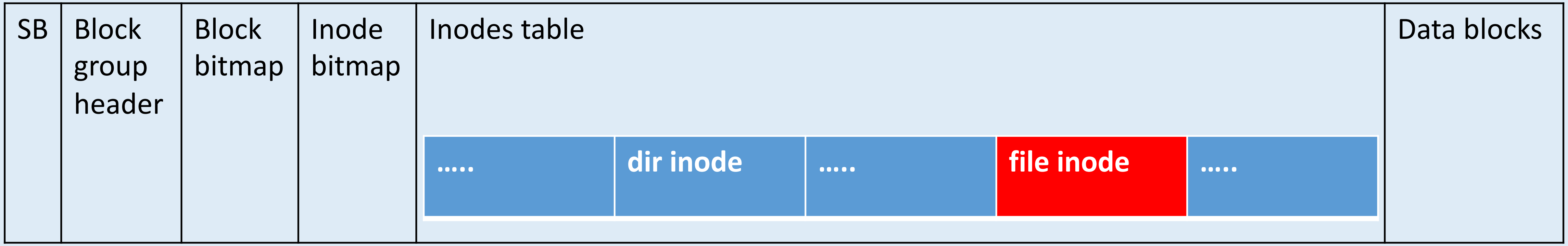
Imagine your machine crash when creating a file

Creating a file is a simple operation from the point of view of a user:

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

A file system, on the other hand, performs multiple steps:

1. Find a free inode,
2. Find free blocks for the file content,
3. Flag the inode and the data blocks as in-use,
4. Fill the inode,
5. Append a struct ext2_dir_entry to the parent directory of the new file,
6. Since the directory length has grown, the FS has to update the inode of the directory.



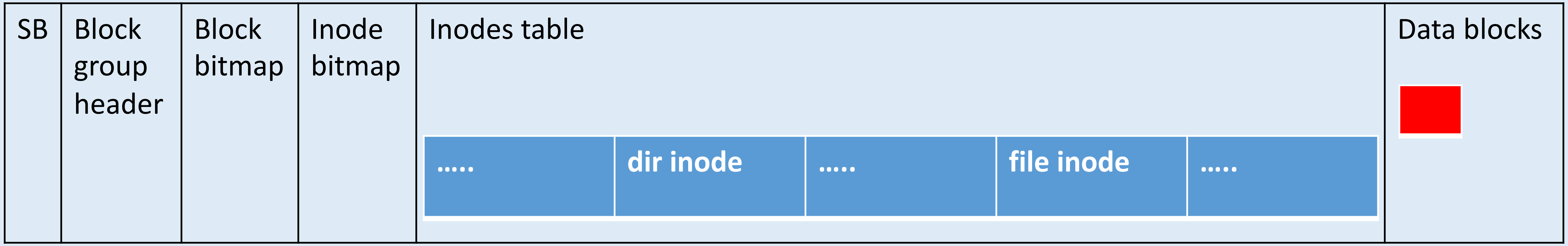
Imagine your machine crash when creating a file

Creating a file is a simple operation from the point of view of a user:

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

A file system, on the other hand, performs multiple steps:

1. Find a free inode,
2. Find free blocks for the file content,
3. Flag the inode and the data blocks as in-use,
4. Fill the inode,
5. Append a struct ext2_dir_entry to the parent directory of the new file,
6. Since the directory length has grown, the FS has to update the inode of the directory.



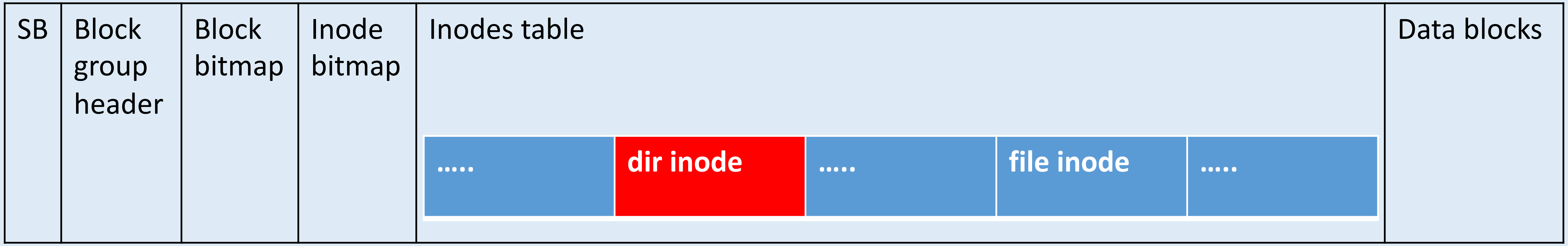
Imagine your machine crash when creating a file

Creating a file is a simple operation from the point of view of a user:

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

A file system, on the other hand, performs multiple steps:

1. Find a free inode,
2. Find free blocks for the file content,
3. Flag the inode and the data blocks as in-use,
4. Fill the inode,
5. Append a struct ext2_dir_entry to the parent directory of the new file,
6. Since the directory length has grown, the FS has to update the inode of the directory.



Imagine your machine crash when creating a file

A file system, on the other hand, performs multiple steps:

1. Find a free inode,
2. Find free blocks for the file content,
3. Flag the inode and the data blocks as in-use,
4. Fill the inode,

   The machine crashed here (e.g. the power was switched off)

5. Append a struct ext2_dir_entry to the parent directory of the new file,
6. Since the directory length has grown, the FS has to update the inode of the directory.

Imagine your machine crash when creating a file

A file system, on the other hand, performs multiple steps:

1. Find a free inode,
2. Find free blocks for the file content,
3. Flag the inode and the data blocks as in-use,
4. Fill the inode,

   The machine crashed here (e.g. the power was switched off)

5. Append a struct ext2_dir_entry to the parent directory of the new file,
6. Since the directory length has grown, the FS has to update the inode of the directory.

After this crash:

1. The inode and some data blocks are flagged as in-use,
2. The counters of free inodes and blocks are decreased,
3. There is no new file in the FS.

What do we need from a journal?

1. Updates to the FS need to be transactional (either all changes are applied, or none at all),
2. Updates to the FS need to preserve the order of operations.

What do we need from a journal?

1. Updates to the FS need to be transactional (either all changes are applied, or none at all),
2. Updates to the FS need to preserve the order of operations.

Why is it important to preserve the order:

PostgreSQL, git and many other applications have data files and metadata files. They first update data files, and then update metadata files to point to new data files.

Reminder: we’ve discussed the split into data and metadata when discussing `fsync()`.

If the system crashes, the FS may not allow the following situation:

- a metadata file was updated, but data files stayed at their previous version.

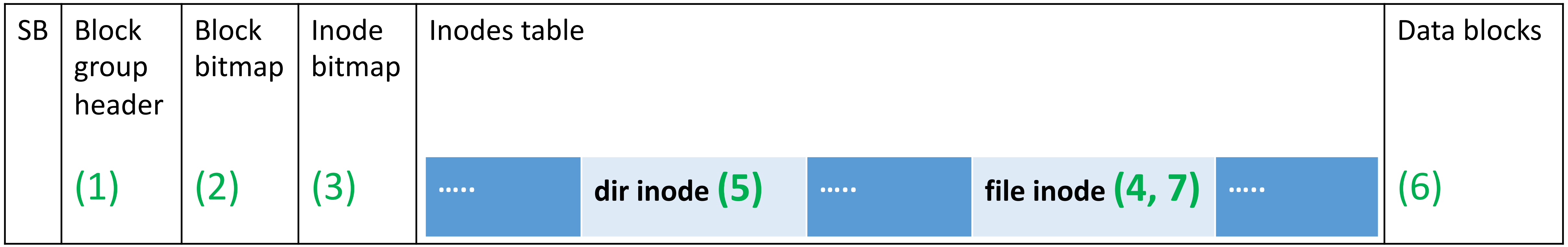
More generally, if an operation by a program is visible after a crash, all operations that *happened-before** it need to be visible, too.

* https://lamport.azurewebsites.net/pubs/time-clocks.pdf

Limiting the FS inconsistencies and soft updates

1. Updates to the FS need to be transactional (either all changes are applied, or none at all),
2. Updates to the FS need to preserve the order of operations.

We may use `fsync()`s or IO barriers* to order the updates to the FS this way:

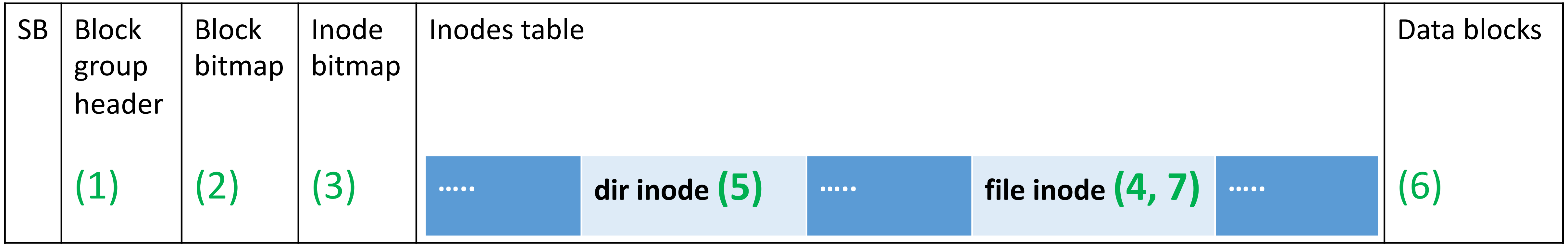


Quiz: *What are read and write memory barriers?*
Quiz: *What is a compiler barrier? How does one request a compiler barrier in GCC and clang?*

Limiting the FS inconsistencies and soft updates

1. Updates to the FS need to be transactional (either all changes are applied, or none at all),
2. Updates to the FS need to preserve the order of operations.

We may use `fsync()`s or IO barriers to order the updates to the FS this way:



In this case we know what kinds of inconsistencies may appear on the disk, and fix them.

See also: Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem, https://users.ece.cmu.edu/~ganger/papers/mckusick99.pdf

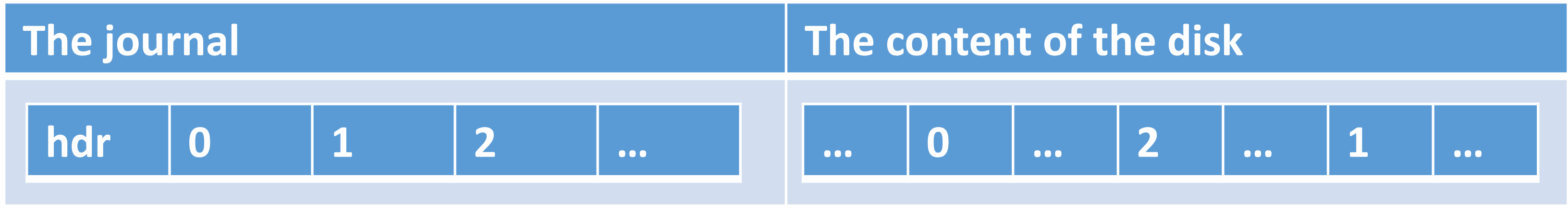
See also: Barrier-Enabled IO Stack for Flash Storage
https://www.usenix.org/system/files/conference/fast18/fast18-won.pdf

What do we need from a journal?

1. Updates to the FS need to be transactional (either all changes are applied, or none at all),
2. Updates to the FS need to preserve the order of operations.

An implementation:

1. Log* blocks to be updated into the journal.
2. `fsync()` to persist the journal.
3. Flush blocks to the disk in the background.
4. Once blocks are flushed to the disk, add an entry to the journal to mark the transaction as completed.



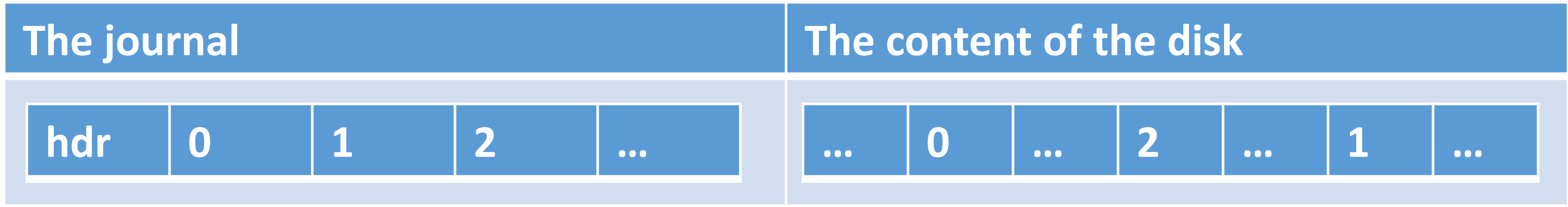
Note: we assume this operation is atomic. We will see later how to implement atomic writes to journal.

What do we need from a journal?

1. Updates to the FS need to be transactional (either all changes are applied, or none at all),
2. Updates to the FS need to preserve the order of operations.

An implementation:

1. Log blocks to be updated into the journal.
2. `fsync()` to persist the journal.
3. Flush blocks to the disk in the background.
4. Once blocks are flushed to the disk, add an entry to the journal to mark the transaction as completed.



If the system crashes while modifying the file system then all changes to apply to the FS stay in the journal. When mounting the FS the next time, we read the journal and apply changes that were not applied because of the crash.

This procedure is called **crash recovery**.

What changes do we journal? Consistent FS state.

Journaling, if implemented as in the previous slide, doubles the number of blocks to write for each change.

What changes do we journal? Consistent FS state.

Journaling, if implemented as in the previous slide, doubles the number of blocks to write for each change.

It is moderately bad performance-wise because writes to the journal are sequential*.

Still, we would like to journal fewer writes.

Quiz: why was this reasoning OK for rotating media, and why is it absurd for SSDs?

What changes do we journal? Consistent FS state.

Journaling, if implemented as in the previous slide, doubles the number of blocks to write for each change.

It is moderately bad performance-wise because writes to the journal are sequential.

Still, we would like to journal fewer writes.

An idea:

- Let us journal only changes to the FS metadata.

This way we will have a file system that is always consistent. It will have no lost blocks or inodes, no directory entries that point to nowhere, etc.

But: there are no guarantees with respect to the application data in files.

What changes do we journal? Consistent FS state.

Let us consider an application that appends some data to a file located on an XFS volume.

XFS will
- Find free zero blocks,
- Journal updates to the block bitmap, to the inode and to the extent tree,
- Write data blocks directly to the disk, bypassing the journal.

Only changes to the FS metadata will be journaled. They are only several kilobytes long.
Unlike that, application data may be gigabytes long. All that data will bypass the journal.

What happens if the system crashes during such append?

What changes do we journal? Consistent FS state.

Let us consider an application that appends some data to a file located on an XFS volume.

XFS will
- Find free zero blocks,
- Journal updates to the block bitmap, to the inode and to the extent tree,
- Write data blocks directly to the disk, bypassing the journal.

Only changes to the FS metadata will be journaled. They are only several kilobytes long.
Unlike that, application data may be gigabytes long. All that data will bypass the journal.

What happens if the system crashes during such append?

As the result, we will have a file which has grown and contains garbage at the tail.

Quiz: describe what kinds of “garbage” are possible.

A reminder: page cache, writeback and `fsync()` failures

`fsync()` and `fdatasync()`

- report whether a writeback succeeded or failed,
- do **not** specify ranges that were written successfully and ranges that failed*.

How do we design our file formats so that an error “some (unspecified) writes failed” can be handled?

Quiz: if a program overwrites a region in a file, what can we assume about the region if `fsync()` fails?

What changes do we journal? Consistent FS state.

Let us consider an application that appends some data to a file located on an XFS volume.

XFS will

- Find free zero blocks,
- Journal updates to the block bitmap, to the inode and to the extent tree,
- Write data blocks directly to the disk, bypassing the journal.

Only changes to the FS metadata will be journaled. They are only several kilobytes long.
Unlike that, application data may be gigabytes long. All that data will bypass the journal.

What happens if the system crashes during such append?

As the result, we will have a file which has grown and contains garbage at the tail.

Application already handle this by organising their storage into data and metadata files, and replacing metadata files atomically after a change to the data is persisted.
Thus, less journaling does not add new failure modes for applications to support.

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

Checkpointing

A journal has a limited number of entries. It needs to be cleared regularly.

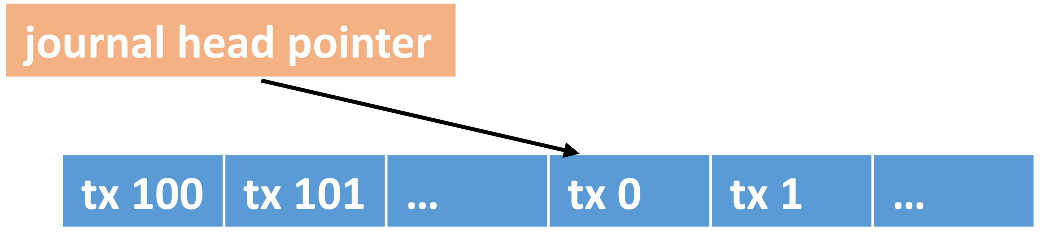
Checkpointing is the process of forcefully applying (some) changes from the journal:

- Apply journaled writes from the head of the journal,
- `fsync()`,
- Advance the pointer to the head of the journal, thus reclaiming journal space.

Checkpointing may run periodically, or when the number of journal entries becomes large enough.

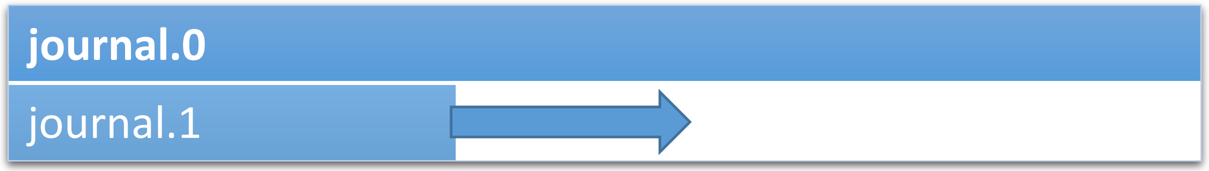
Two ways to implement a journal

As a ring buffer



When the journal overflows, we wrap to the start of it. We need to be careful and not overwrite transaction that are not yet committed.

As multiple files
(this is suitable for applications, but not the FS itself)

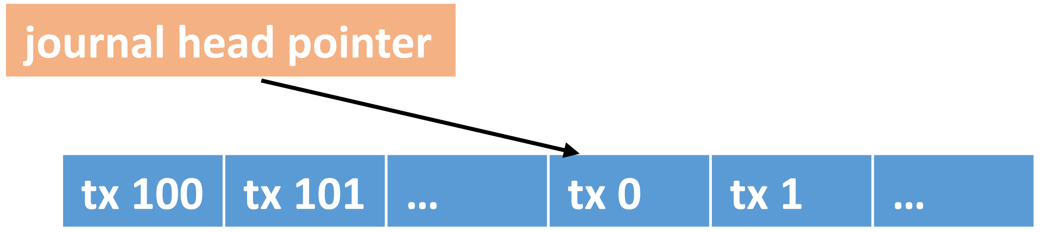


When a file overflows we just continue writing a higher-numbered one.

As we commit the transaction we remove older journals that have been fully replayed.

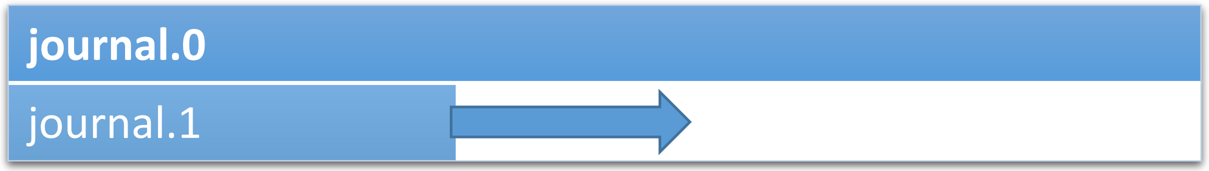
Two ways to implement a journal

As a ring buffer



When the journal overflows, we wrap to the start of it. We need to be careful and not overwrite transaction that are not yet committed.

As multiple files
(this is suitable for applications, but not the FS itself)



When a file overflows we just continue writing a higher-numbered one.

As we commit the transaction we remove older journals that have been fully replayed.

Remark: A simple way is to append records to the tail of the journal. That is not efficient. A much better implementation is to reserve the space for a journal file, and never grow it. In that case `fdatasync()` instead of `fsync()` will suffice.

** Why the pair of `fallocate() + fdatasync()` can be several times faster than `fsync()`?*

What operations to log?

Logical changes to the FS:

- Creation/removal/renaming of files,
- Resizing of files,
- Changes to access rights,
- …

Physical changes to the FS: the content of blocks on the disk to overwrite with updated structures.

What operations to log?

Logical changes to the FS:

- Creation/removal/renaming of files,
- Resizing of files,
- Changes to access rights,
- …

Physical changes to the FS: the content of blocks on the disk to overwrite with updated structures.

A journal of logical changes may be much smaller than a journal of physical changes. For example, an ext4 inode is 256 bytes long. An update to an inode needs to log only 256 bytes. An update to a block that contains the inode needs* to log 4k worth of data.

Remark: XFS logs overwrites of disk regions at much finer granularity to avoid this.

What operations to log?

Logical changes to the FS:

- Creation/removal/renaming of files,
- Resizing of files,
- Changes to access rights,
- …

Physical changes to the FS: the content of blocks on the disk to overwrite with updated structures.

What will happen if the system crashes while replaying the journal and we need to replay it again upon the next mount?

What operations to log?

Logical changes to the FS:

- Creation/removal/renaming of files,
- Resizing of files,
- Changes to access rights,
- …

Physical changes to the FS: the content of blocks on the disk to overwrite with updated structures.

What will happen if the system crashes while replaying the journal and we need to replay it again upon the next mount?

In general, transactions can’t be replayed twice: how does one repeat `rename(“a”, “b”)`?

Operations “overwrite a block N” are **idempotent**. Replaying them many times has the same effect as replaying them once.

Case study: idempotent operations

The idempotency of operations is also important when we design networking protocols.

A request sent over the Internet may be lost. In this case our only recourse is to retry a request.

The unreliability of the network introduces new failure modes that need to be handled:

1. A server may receive the same request twice (the original one and the retry).
2. If a request times out, a client cannot know whether the request was never processed, or it succeeded, or it failed, or it is still being handled.

Case study: idempotent operations

Consider the following protocol of a networked storage:

```
• Open:      (file_name, lock_level)            -> lock_id,
• Read:      (file_name, lock_id, offset, size) -> data,
• Append:    (file_name, lock_id, offset, data) -> (),
• PunchHole: (file_name, lock_id, offset, len)  -> (),
• Close:     (lock_id)                          -> (),
• Rename:    (file_name_src, file_name_dst)     -> (),
• SwapFiles: (file_path_0, file_path_1)         -> ().
```

Possible values of `lock_level` are:

- Shared (reader locks),
- Normal (there may be only 1 normal writer, and it allows concurrent shared readers),
- Exclusive (there may be only 1 exclusive writer, and no other users of the file).

Quiz: What problems does this protocol have?
Hint: requests are sent over an unreliable network. What happens if a request needs to be retried?

Case study: idempotent operations

Amazon EC2 has the following API to create VMs:

```
type RunInstancesInput struct {
    ...
    ClientToken         *string
    ...
    BlockDeviceMappings []*BlockDeviceMapping
    CpuOptions          *CpuOptionsRequest
    ImageId             *string
    ...
}
```

The parameter `ClientToken` is a random string generated by the client. If EC2 receives two requests with the same `ClientToken`, it knows that the second one is a retry. It does not create a second VM, but returns the result of the first call.

Quiz: how do we fix `Open()` from the previous slide to become retryable?

Reordering in the network

The idempotency of operations is also important when we design networking protocols.

A request sent over the Internet may be lost. In this case our only recourse is to retry a request.

The unreliability of the network introduces new failure modes that need to be handled:

1. A server may receive the same request twice (the original one and the retry).
2. If a request times out, a client cannot know whether the request was never processed, or it succeeded, or it failed, or it is still being handled.

Reordering in the network

The idempotency of operations is also important when we design networking protocols.

A request sent over the Internet may be lost. In this case our only recourse is to retry a request.

The unreliability of the network introduces new failure modes that need to be handled:

1. A server may receive the same request twice (the original one and the retry).
2. If a request times out, a client cannot know whether the request was never processed, or it succeeded, or it failed, or it is still being handled.
3. The original request may arrive to the server after a retried one.

Reordering in the network

The idempotency of operations is also important when we design networking protocols.

A request sent over the Internet may be lost. In this case our only recourse is to retry a request.

The unreliability of the network introduces new failure modes that need to be handled:

1. A server may receive the same request twice (the original one and the retry).
2. If a request times out, a client cannot know whether the request was never processed, or it succeeded, or it failed, or it is still being handled.
3. The original request may arrive to the server after a retried one.

How can such reordering happen? It is a rather typical scenario in virtualised environments like Kubernetes. Normally, Kubernetes nodes are VMs. They may be migrated, and a migration failure can produce this:

1. A VM migration begins which renders the node N unavailable.
2. A migration gets stuck and cannot restore N quickly enough.
3. Kubernetes detects the unavailability of N. A process P that was running on N gets rescheduled to run another node N’. This will be a process P’.
4. P’ makes updates to a database and terminates.
5. A migration completes and the node N returns to the cluster. The process P resumes.
6. P issues updates to the (already updated) database.

Reordering in the network

Google Cloud Storage assigns a generation to each object, and PUT Object has two versions:

1. curl -X PUT https://storage.googleapis.com/${BUCKET}/${OBJECT}
2. curl -X PUT https://storage.googleapis.com/${BUCKET}/${OBJECT} -H "x-goog-if-generation-match: ${GENERATION}”

The first version replaces an object unconditionally.

The second version replaces an object only if it has an expected value of the generation number.

Reordering in the network

Google Cloud Storage assigns a generation to each object, and PUT Object has two versions:

1. curl -X PUT https://storage.googleapis.com/${BUCKET}/${OBJECT}
2. curl -X PUT https://storage.googleapis.com/${BUCKET}/${OBJECT} -H "x-goog-if-generation-match: ${GENERATION}”

The first version replaces an object unconditionally.

The second version replaces an object only if it has an expected value of the generation number.

There are more applications of “assume success and abort when a locking conflict is detected”. Postgres uses it to implement serialisable transactions: https://github.com/postgres/postgres/blob/master/src/backend/storage/lmgr/README-SSI.

Atomicity of journal writes

Writing multiple sectors to a disk is not atomic. If a record in the journal takes multiple sectors, how do we guarantee that it is written in full?

Atomicity of journal writes

Writing multiple sectors to a disk is not atomic. If a record in the journal takes multiple sectors, how do we guarantee that it is written in full?

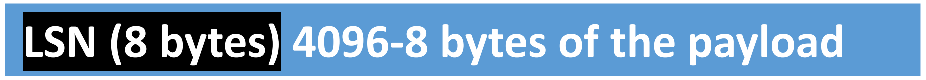
Writing a single sector to a disk is atomic. The word “atomic”, however, has a different meaning than in “atomic memory write”. After a write a sector may

1. be overwritten fully,
2. remain unchanged,
3. become corrupted and return IO errors until it is overwritten.

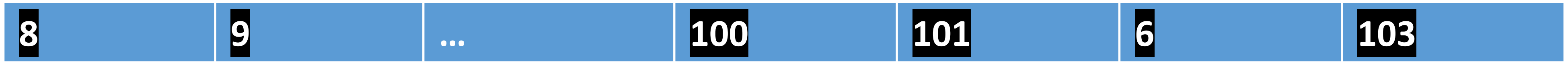
Atomicity of journal writes

Writing multiple sectors to a disk is not atomic. If a record in the journal takes multiple sectors, how do we guarantee that it is written in full?

Starts all sectors of the journal with a Log Sequence Number:



Log Sequence Number grows monotonically:



Transactions to be replayed can be found by locating ranges of sectors with consecutive LSNs (exercise: fill in the details).