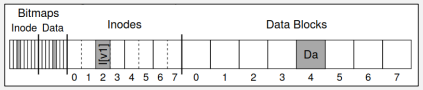
A Problem

- assume a simple (and tiny) file system
  - inode bitmap (8 bits)
  - data bitmap (8 bits)
  - 8 inodes (4 blocks)
  - 8 data blocks

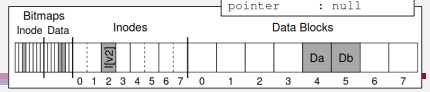


- setup: there is one single-block file (inode 2, data block 4)

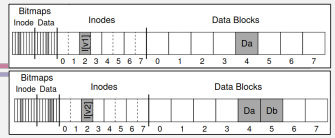
```
owner       : remzi
permissions : read-write
size        : 1
pointer     : 4
pointer     : null
pointer     : null
pointer     : null
```

- task: append one data block to this file
  - requires three writes: inode, bitmaps, data block
  - writes are typically cached (e.g. 5-30s)

```
owner       : remzi
permissions : read-write
size        : 2
pointer     : 4
pointer     : 5
pointer     : null
pointer     : null
```

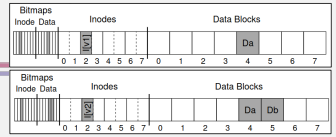


Crash Scenarios



- only the data block is written
  - data is on disk, but nothing refers to it – not inconsistent, though data is lost
- only the updated inode is written
  - inode references a data block that hasn't been written – *garbage* data
  - *file-system inconsistency* – inode says data block is allocated but data bitmap says not
- only the updated bitmap is written
  - *space leak* – data bitmap says data block is allocated but inode doesn't refer to it (data block will never be used)

Crash Scenarios



- only the inode and bitmap are written
  - garbage data – file system is consistent (both inode and data bitmap think the data block is allocated) but the data block contains junk
- only the inode and data block are written
  - *file-system inconsistency* – inode says data block is allocated but data bitmap says not
- only the bitmap and data block are written
  - data bitmap says data block is allocated but inode doesn't refer to it – data is there, but which file it belongs to has been lost

Crash Consistency

> THE CRUX: HOW TO UPDATE THE DISK DESPITE CRASHES
> The system may crash or lose power between any two writes, and thus the on-disk state may only partially get updated. After the crash, the system boots and wishes to mount the file system again (in order to access files and such). Given that crashes can occur at arbitrary points in time, how do we ensure the file system keeps the on-disk image in a reasonable state?

- strategies
  - file system checker (fsck)
  - journaling (write-ahead logging)
  - something else

`fsck`

- strategy: fix any file system inconsistencies on boot (before file system is mounted)
  - only concerned with file system consistency – doesn't address garbage data

`fsck` Tasks

- check superblock
  - performs sanity checks (e.g. file system size > number of allocated blocks) to detect corrupt superblock
- check free blocks
  - scan inodes to determine which blocks are currently allocated
  - resolve inconsistencies between inodes and bitmaps in favor of the inodes
- check inode state
  - if allocated inodes have invalid types or other problems not easily fixed, the inode is cleared
- check inode links
  - scan directory tree to compute link count for all files and directories
  - update count in inode if that is incorrect; move allocated inodes not referred to by a directory to lost+found directory
- check for duplicate pointers
  - if two different inodes refer to the same block, clear bad inode or copy the pointed-to block
- check for bad blocks
  - "bad" pointer refers outside of the valid range – clear from inode
- check directories
  - integrity checks e.g. `.` and `..` are first two entries, referenced inodes are allocated, no directory is linked to more than once

`fsck` Challenges

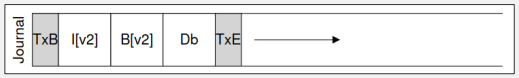
- complex, and can be tricky to handle all edges cases
- very slow, and becomes impractical for large disks
  - also overkill to check entire disk when the likelihood is that only a very small number of problems might exist

Journaling

- used by ext3, ext4 (Linux), NTFS (Windows)
- *write-ahead logging*
  - log the actions about to be taken in a well-known location on disk
  - fixing problems only requires consulting the log
  - adds overhead to writes in order to speed up recovery

ext3 adds the journal to each block group

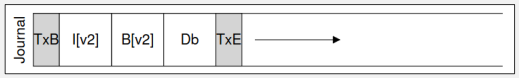
Data Journaling in ext3



- setup: there is one single-block file
- task: append one data block to this file

- journal entry consists of –
  - *transaction begin* (TxB) – includes info about the update (e.g. addresses of the blocks to be written), transaction identifier (TID)
  - the three blocks to be written (*physical logging*)
    - *logical logging* of the task ("append data block Db to file X") is more compact
  - *transaction end* (TxE) – includes TID

Data Journaling in ext3



- sequence of operations
  - *journal write* – write the journal entry (except TxE) and wait for the writes to complete
  - *journal commit* – write TxE and wait for the write to complete
    - requires atomicity of writing a single sector to the disk (make TxE exactly that size)
    - two-stage write of journal entry balances efficiency of larger writes with being able to detect an incomplete entry if there is a crash while writing the entry
    - ext4 instead writes entire entry at once along with a checksum – allows for detection of incomplete entry on recovery
  - *checkpoint* – write the actual updates

ensuring writes have completed is complicated by write buffering on disks – the disk may report write complete while it is still buffered

Journaling – Recovery

- if a crash happens –
  - during the write of the journal entry (before journal commit)
    - detected by missing TxE or invalid checksum
    - action: discard the pending update
      - no changes have been made to the filesystem so no inconsistency
  - after journal commit but before checkpoint is complete
    - *redo logging* – to recover the update on boot, scan the log and replay the committed transactions
      - at worst, some updates are repeated

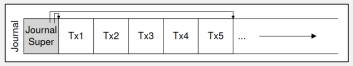
Two Wrinkles

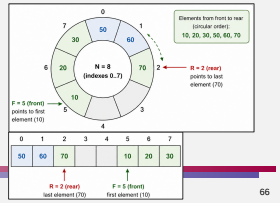
- a single update can involve changes to many blocks
  - e.g. create file involves the inode bitmap, the file's inode, the parent directory's inode, the parent directory's data block
- a sequence of updates can involve multiple changes to the same blocks
  - e.g. creating two files in the same directory means two changes to the inode bitmap, the parent directory's inode, the parent directory's data block

- simply adding to the log with each write is problematic
  - a larger log means recovery takes longer
  - eventually the space available for the log fills up

Two Solutions

- buffer updates in memory and *batch* log entries
  - maintain in-memory copies of the affected disk blocks
  - periodically write a collection of updates to disk as a single transaction
- free log space once a transaction has been checkpointed
  - treating the log as a *circular log* avoids having to shift entries
  - *journal superblock* records the oldest and newest non-checkpointed transactions





Data Journaling Recap

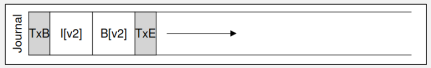
1. **Journal write:** Write the contents of the transaction (containing TxB and the contents of the update) to the log; wait for these writes to complete.
2. **Journal commit:** Write the transaction commit block (containing TxE) to the log; wait for the write to complete; the transaction is now **committed**.
3. **Checkpoint:** Write the contents of the update to their final locations within the file system.
4. **Free:** Some time later, mark the transaction free in the journal by updating the journal superblock.

improved `ext4` version: write transaction along with a checksum in a single step, then check checksum on replay

- recovery is fast – just replay un-checkpointed entries
- routine is slow – blocks are written twice (once for real, once as part of the journal entry) and in different parts of the disk (requiring a seek)
  - this is expensive for a routine practice, especially when the problem being solved is rare

Metadata Journaling

- don't write the data blocks to the journal



1. **Data write:** Write data to final location; wait for completion (the wait is optional; see below for details).
2. **Journal metadata write:** Write the begin block and metadata to the log; wait for writes to complete.
3. **Journal commit:** Write the transaction commit block (containing TxE) to the log; wait for the write to complete; the transaction (including data) is now **committed**.
4. **Checkpoint metadata:** Write the contents of the metadata update to their final locations within the file system.
5. **Free:** Later, mark the transaction free in journal superblock.

write data blocks first to avoid garbage data

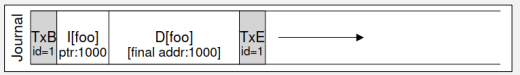
(no inconsistency problems if data blocks are written but inodes are not, but the reverse results in garbage data)

can write data and journal metadata concurrently – only need steps 1-2 to finish before step 3

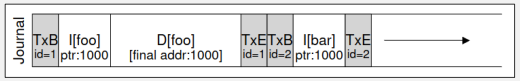
The Challenge of Deletion

- create a file in directory `foo`

assume metadata journaling



- delete the directory `foo` and all of its contents
- create a new file `bar`
  - (which happens to be allocated the now-freed block formally used by `foo`)



with metadata journaling, the data block for `bar` is not written to the journal – so when the journal is replayed after a crash, the (now-old) `D[foo]` is written to block 1000 with the inode for `bar` pointing there

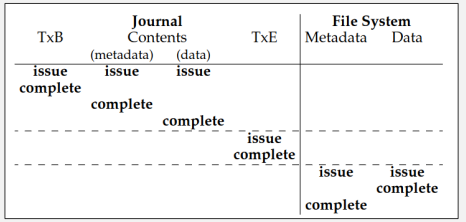
The Challenge of Deletion

- solutions
  - don't reuse blocks until their deletion has been checkpointed
  - (`ext3`) write a *revoke* record to the journal on deletion, then on recovery scan the log for revoke records before replaying

Journaling Timelines

dashed lines are write barriers – every before must complete before anything after
otherwise ordering of completions is arbitrary – writes may be reordered to improve performance

data journaling



metadata journaling