CS 423
Operating System Design: Journaled File Systems & LFS
Apr 21

Ram Kesavan

LOGISTICS

MP3 grades + breakdown released

- Resubmission deadline Apr 27, 11:59 CT
- Only if you did not resubmit MP2

Midterm papers: pickup from my office during OH

MP4 due by May 6, 11:59 CT

Last lecture: 5/5 will be a review

Finals: 5/13 at 7-10pm, DCL 1320 (next door)

AGENDA / LEARNING OUTCOMES

Crash Consistency

- Fsck
- Journaling

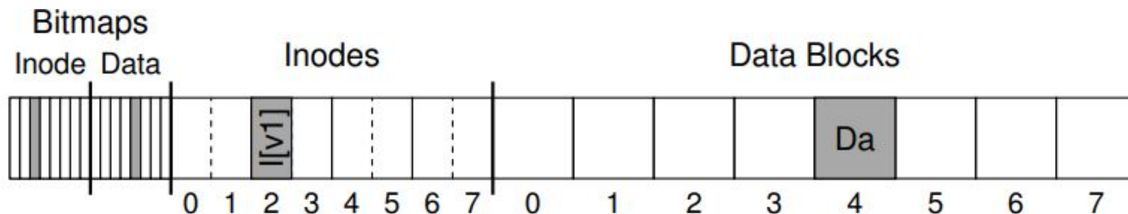
Log-structured File System (LFS)

- Journal is the file system!

RECAP

Append a Block Example

Append 5 bytes to a file, which adds a new block



How many blocks need to be updated to accomplish the append? And, which ones? Assume: overflows 4KiB boundary

1. Allocate a new block. Add pointer to this new block–either new direct block pointer in inode (if eventual size is <= 48 KiB), or in one of the indirect blocks (single or double). Allocate new indirect block(s) if eventual size extends past the ~4MiB, ~4GiB, ~4TiB boundary. Each newly allocated block requires a bit to be flipped in the datablock-bitmap.
2. The actual data to the last 2 data blocks of the file.
3. Update inode's metadata: mtime, size, block pointer, and #blocks.

A Working File System

user space
app

(1) fd = open(file, ..)
(3) R(fd, offset, 4KB)
(6) W(fd, offset+4086, 20)

kernel
inode cache
fd
i
ptr
size
mtime
#blks
buffer cache
buf
0->1
jth block
storage
S
ib
db
inodes
data
j

Ack (11): the write has been persisted

Ack (9): the write has not been persisted

FSCK

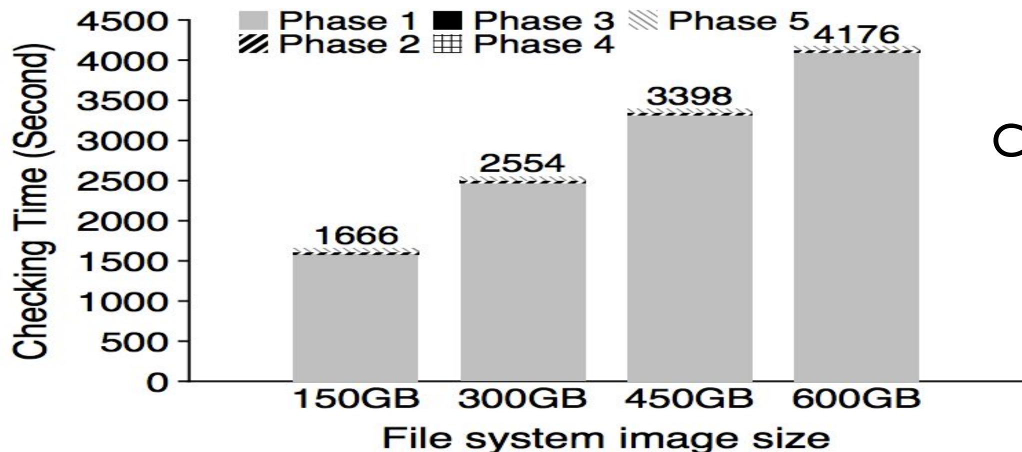
Let inconsistencies happen; fix up during reboot
File system check, aka fsck (pronounced fisk)

```
UNEXPECTED SOFT UPDATE INCONSISTENCY
** Last Mounted on /
** Root file system
** Phase 1 - Check Blocks and Sizes
** Phase 2 - Check Pathnames
** Phase 3 - Check Connectivity
** Phase 4 - Check Reference Counts
UNREF FILE I=9470237  OWNER=mysql MODE=100600
SIZE=0 MTIME=Feb  9 06:52 2016

CLEAR? no

** Phase 5 - Check Cyl groups
FREE BLK COUNT(S) WRONG IN SUPERBLK
SALVAGE? no

SUMMARY INFORMATION BAD
SALVAGE? no

BLK(S) MISSING IN BIT MAPS
SALVAGE? no

722171 files, 11174866 used, 8118876 free (156260 frags, 995327 blocks, 0.8% fra
gmentation)
\[\033[01;34m\]root@\[\033[00m\]:\[\033[01;34m\]/\[\033[00m\]#
```

END RECAP

Fsck Problems

- Not always obvious how to fix file system image
- A consistent state isn't necessarily the "correct" state
- Updates could be lost, not really an option for some applications (eg. database)
- Simply too slow!



Checking a 600GB disk takes ~70 minutes

ffsck: The Fast File System Checker
Ao Ma, Chris Dragga, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau

Journaling or WAL

WAL: write ahead logging

Foundational database technique for “modify” ops

Basic idea:

- write a “note” to a well-known location about the op
  - while modifying in-memory file system
- once “note” is persisted
  - ack the op to the client
  - then, modify the persistent file system (aka checkpointing)
- if crash, replay all notes on restart for fast repair
  - avoid an fsck-style full file-system walk

Tradeoff: additional work during op→reduce work at restart & **full** recovery

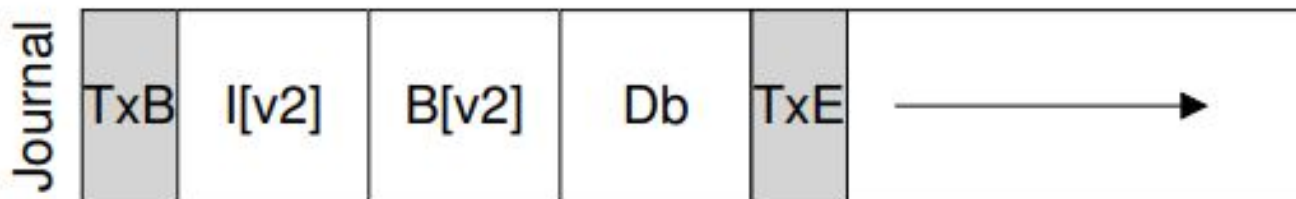
Note: fsck avoided on restart, but still needed when file system corruption occurs!

Data Journaling in Linux ext3

Journal: sequence of journal entries (one per op)

- Book-end each journal entry (TxB & TxE)
- Entry is a list of blocks that need to be persisted: aka **physical journaling**
- TxB metadata: ID, length of entry, block# location where each block is to be persisted

Journal entry for file append that adds a block



Finite Size of Journal

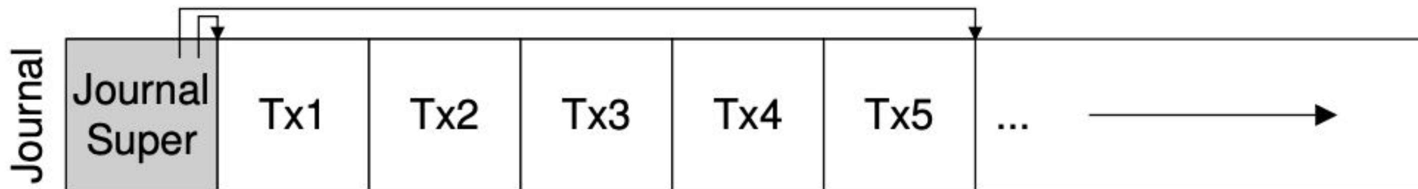
Once checkpointed a journal entry can be reclaimed

- Else, replay is unnecessarily slower & journal overflows

Treat journal as a circular log

In journal header, store pointers:

- head: oldest non-checkpointed journal entry
- tail: newest non-checkpointed journal entry



Journaling

Journal: each “modify” op gets a journal entry

Journal performance: optimized for append writes

Where: 2 options for journal location

- Within the file system (ext3) and/or same storage device as the file system
- Separate storage device

How: to tell which entries to replay after a crash?

- Journal header head pointer
- Periodically update superblock with the most-recently checkpointed journal entry ID

What: to journal: 2 options

- Physical logging: all blocks resulting from the op: optimized for fast replay
- Op (aka Logical) logging: minimum info about the op: optimized for runtime (journaling) speed

Discuss with previous eg: append 5 bytes to a file, which adds a block to the file

Journaling: What

Previous eg: append 5 bytes to a file, which adds a block to the file

Size of journal entry:

- **Physical** (12+ KiB): at least 1 data block, 1 bitmap block, 1 inode block, maybe indirect blocks
- **Op/Logical** (17 bytes): write, inum, offset, data: 4 + 4 + 4 + 5

Physical logging:

- Journal needs more storage space & bandwidth
- Wasteful: blocks are written twice, to journal and then to file system
    - Option: metadata-only (data is not logged); possible for some applications

Op/Logical Logging: requires replay idempotence

In our eg: append replay must persist to the same block numbers
Else, what bad thing could happen?

Journaling & Checkpointing

Each "modify" op can be thought of a transaction
- Atomic (like to a database)
- Each log/journal entry is for a single transaction

Once a log-entry has been persisted
- (In-memory file system should've already been modified–aka "dirtied")
- ack the operation to the client
- **checkpoint**: write all changes (dirty in-memory state) to the persistent file system
  - then reclaim the log-entry & mark in-memory state clean

What if crash happens during the:
- Checkpointing of a transaction? Redo logging
- Write of a journal entry?

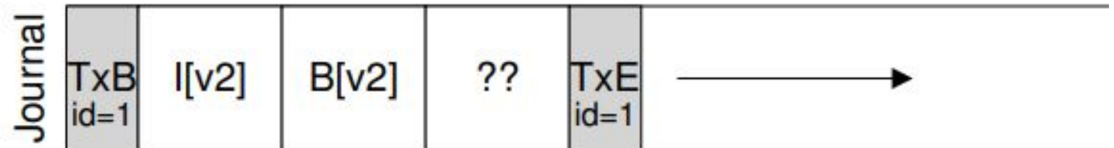
Maximize write throughput:
- To journal: issue writes in parallel to all blocks of an entry
- To file system: issue writes in parallel to all blocks being updated in a checkpoint

Journal Writes: Ordering?

Checkpoint to Filesystem: no ordering (between writes) necessary
- Just wait for all writes to complete
- If crash before that, journal replay idempotency guarantees correctness

Write to journal entry: Can issue all writes except TxE
- Wait for all writes to complete
- Write TxE & wait for that to complete
  - TxE commits the journal entry
- Otherwise, replay can't tell if the log entry is complete

Can we do better? (ext4)



Write Throughput

What is the primary performance bottleneck with journaling?

Write-throughput of the Journal

- Store journal on high-throughput device; eg. Non-volatile RAM (NVRAM)
- Use op/logical logging
- Metadata/ordered logging

Write-throughput of persistent file systems

- Say the user app creates a bunch of files in a directory
- Same blocks need to written over and over again
  - Dir data block(s), dir inode
  - Inode block(s), inode bitmap

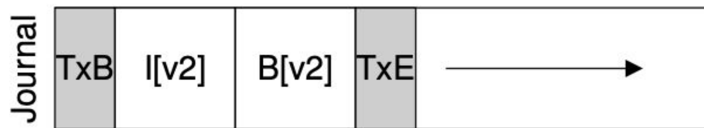
Metadata Journaling

Bulk of blocks written to journal are user file data
Don't journal the user file data blocks!

- aka ordered journaling; eg. ext3
- write user file data blocks to file system directly

Ordering implications?

- checkpoint data blocks directly to file system
  - even before transaction is written to journal!
  - guarantees that no metadata points to garbage
- ensure all data blocks are persisted before (metadata of) op is checkpointed
- maximizes parallelism for writes to file system



Performance Tradeoff

Delay vs don't-delay the checkpointing process

Early checkpoint of (the metadata of) op too!

Don't-delay:

Pro: can reclaim journal entries sooner
Pro: recovery (after a restart) replays fewer journal entries
Con: undo logging will be needed; why?

Delay:

Con: journal needs more space
Pro: batching updates to metadata amortizes that write overhead
Pro: writing large #blks together is more efficient

Redo & Undo Logging

Choice

1. T4 ckpting starts AFTER T4 is committed to journal
2. T4 ckpting starts BEFORE T4 is committed

RECOVERY

1. REDO T2, T3
2. REDO T2, T3; UNDO T4

Tradeoff: ② has to log more stuff (to be undone) so it can start ckpting early.

in-memory fs

T1 (clean) — done

T2 (dirty) — ckpt'ing

T3 (dirty)

T4 (dirty) — option 2 — log writing

Log: ⋮, T1, T2, T3, T4, …

ckpt'ed

committed

current

persistent fs

Redo & Undo Logging

3 pointers:

- ckpt’ed: all prior entries are checkpointed and/or reclaimed
- committed: all prior entries are fully committed to journal
- current: the next write to journal

Log Replay

- Redo logging
  - All entries from ckpt’ed to committed
- Undo logging (if checkpointing is allowed before journal entry is committed)
  - All entries from committed to current

Note: Redo log replay has to complete before undo log replay

Large Checkpoint

Most file systems: delay checkpoint to file system

- Avoid writing multiple copies of the same metadata blocks
- Accumulate dirty in-memory state (inodes & buffers)
- **Trade-off**: longer the delay, better the amortization but larger memory footprint of dirty state
- Checkpoint many transactions together

Concept of a *compound transaction*

- Journal several ops as one large compound transaction
- TxB followed by many
  - descriptor blks: details which blks are being updated
  - modified metadata and data blocks
  - terminated by TxE

Log-structed file systems (LFS): the journal is the file system!