File System Robustness

- 11H File System Robustness
- 11I Check-sums
- 11J Log-Structured File Systems

Persistence vs Consistency

- Posix Read-after-Write Consistency
  - any read will see all prior writes
  - even if it is not the same open file instance
- Flush-on-Close Persistence
  - *write(2)* is not persistent until *close(2)* or *fsync(2)*
  - think of these as *commit* operations
  - *close(2)* might take a moderately long time
- This is a compromise …
  - strong consistency for multi-process applications
  - enhanced performance from write-back cache

File Systems - Device Failures

- Unrecoverable Read Errors
  - signal degrades beyond ECC ability to correct
  - background *scrubbing* can greatly reduce
- mis-directed or incomplete writes
  - detectable w/independent checksums
- complete mechanical/electronic failures
- all are correctable w/redundant copies
  - mirroring, parity, or erasure coding
  - individual block or whole volume recovery

File Systems – System Failures

- writes that don't get completed
  - queued writes or data in *write-back cache*
  - client writes/creates that will not be persisted
  - partial multi-block file system updates
- cause – power failures
  - solution: NVRAM disk controllers
  - solution: Uninterruptable Power Supply
  - solution: super-caps and fast flush
- cause – system crashes

Deferred Writes – worst case scenario

- process allocates a new block to file A
  - we get a new block (x) from the free list
  - we write out the updated I-node for file A
  - we defer free-list write-back (happens all the time)
- the system crashes, and after it reboots
  - a new process wants a new block for file B
  - we get block x from the (stale) free list
- two different files now contain the same block
  - when file A is written, file B gets corrupted
  - when file B is written, file A gets corrupted

File Systems: What can go wrong?

- data loss
  - file or data is no longer present
  - some/all of data cannot be correctly read back
- file system corruption
  - lost free space
  - references to non-existent files
  - corrupted free-list multiply allocates space
  - file contents over-written by something else
  - corrupted directories make files un-findable
  - corrupted I-nodes lose file info/pointers

Robustness – Ordered Writes

- ordered writes can reduce potential damage
- write out data before writing pointers to it
  - unreferenced objects can be garbage collected
  - pointers to incorrect info are more serious
- write out deallocations before allocations
  - disassociate resources from old files ASAP
  - free list can be corrected by garbage collection
  - shared data is more serious than missing data

Robustness – Audit and Repair

- design file system structures for audit and repair
  - redundant information in multiple distinct places
    - maintain reference counts in each object
    - children have pointers back to their parents
    - transaction logs of all updates
  - all resources can be garbage collected
    - discover and recover unreferenced objects
- audit file system for correctness (prior to mount)
  - all object well formatted
  - all references and free-lists correct and consistent
- use redundant info to enable automatic repair

Practicality – Ordered Writes

- write barriers
  - all writes before barrier complete before any after it
  - disk drives re-order requests to optimize head motion
  - will the drive honor a write-barrier?
- reduced I/O performance
  - eliminates head/disk motion scheduling
  - eliminates accumulation of near-by operations
  - eliminates consolidation of updates to same block
- doesn't actually solve the problem
  - does not eliminate incomplete writes
  - it chooses minor problems over major ones

Practicality - Audit and Repair

- integrity checking a file system after a crash
  - verifying check-sums, reference counts, etc.
  - automatically correct any inconsistencies
  - a standard practice for many years (see *fsck(8)*)
- it is no longer practical
  - check a 2TB FS at 100MB/second = 5.5 hours
- we need more efficient partial write solutions
  - file systems that are immune to them
  - file systems that enable very fast recovery

Journaling

- create circular buffer journaling device
  - journal writes are always sequential
  - journal writes can be batched (e.g. ops or time)
  - journal is relatively small, may use NVRAM
- journal all intended file system updates
  - I-node updates, block write/alloc/free
- efficiently schedule actual file system updates
  - write-back cache, batching, motion-scheduling
- journal completions when real writes happen

Batched Journal Entries

- operation is safe after journal entry persisted
  - caller must wait for this to happen
- small journal writes are still inefficient
  - accumulate batch until full or max wait time

```
writer:
    if there is no current in-memory journal page
        allocate a new page
    add my transaction to the current journal page
    if current journal page is now full
        do the write, await completion
        wake up processes waiting for this page
    else
        start timer, sleep until I/O is done
```

```
flusher:
    while true
        sleep()
        if current-in-memory page is due
            close page to further updates
            do the write, await completion
            wake up processes waiting for page
```

Journal Recovery

- journal is a circular buffer
  - it can be recycled after old ops have completed
  - time-stamps distinguish new entries from old
- after system restart
  - review entire (relatively small) journal
  - note which ops are known to have completed
  - perform all writes not known to have completed
    - data and destination are both in the journal
    - all of these write operations are idempotent
  - truncate journal and resume normal operation

Why Does Journaling Work?

- journal writes much faster than data writes
  - all journal writes are sequential
  - there is no competing head motion
- in normal operation, journal is write-only
  - file system never reads/processes the journal
- scanning the journal on restart is very fast
  - it is very small (compared to the file system)
  - it can read (sequentially) w/huge (efficient) reads
  - all recovery processing is done in memory

Meta-Data Only Journaling

- Why journal meta-data
  - it is small and random (very I/O inefficient)
  - it is integrity-critical (huge potential data loss)
- Why not journal data
  - it is often large and sequential (I/O efficient)
  - it would consume most of journal capacity/bw
  - it is less order sensitive (just precede meta-data)
- Safe meta-data journaling
  - allocate new space, write the data
  - then journal the meta-data updates

Check-sums

- Parity … detecting single-bit errors
  - one bit per block, odd number of 1-bits
- Parity … restoring lost copy
  - one block per N, XOR of the other N blocks
- Error Correcting Codes
  - detect double-bit errors, correct single-bit errors
- Cryptographic Hash Functions
  - very high probability of detecting any change

Simple Data Checksums

- parity and ECC are stored with the data
  - to identify and correct corrupted data
  - controller does encoding, verification, correction
- very effective against single-bit errors
  - which are common in storage and transmission
- strategy: disk scrubbing
  - slow background read of every block on the disk
  - if there is a single-bit error, ECC will correct it
    - before it can turn into a multi-bit error

Higher Level Checksums

- store the checksum separate from the data
  - it can still be used to detect/correct errors
  - it can also detect valid but wrong data
- many levels at which to check-sum
  - I-node stores a list of block check-sums
    - in de-dup file systems, check-sum is block identifier
  - I-node stores check-sum for the entire file
    - if file is corrupted, go to a secondary copy
  - hierarchical check-sums all the way up the tree

Delta Checksum Computation

- a checksum of many blocks is expensive
  - each block must be read and summed
- updating any block requires a new checksum
  - the dumb way
    - re-read and sum every block again
  - the smart way
    - compute checksum(newBlock)-checksum(oldBlock)
    - add that to checksum(allBlocks)
- choose checksum algorithm accordingly

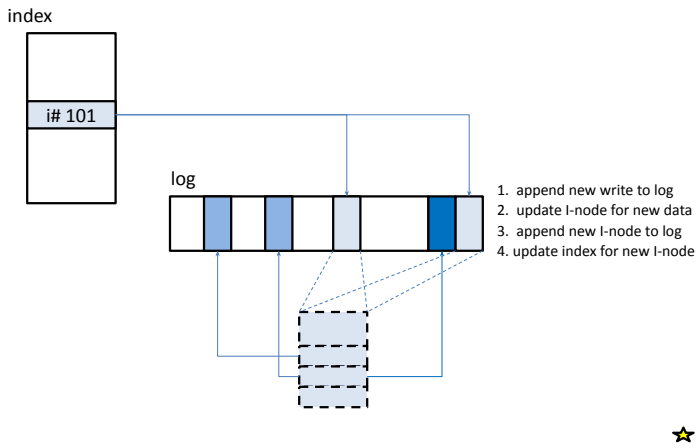
Log Structured File Systems

- the Journal is the file system
  - all I-nodes and data updates written to the log
  - all updates are Redirect-on-Write
  - in-memory index caches I-node locations
- becoming a dominant architecture
  - flash file systems
  - key/value stores
- issues
  - recovery time (to reconstruct index/cache)
  - log defragmentation and garbage collection

Writing/Reading a logging file system

- All writes are appended to the end of the log
  - simple and relatively efficient
- I-nodes point at data segments in the log
  - sequential writes may be contiguous in log
  - random updates will be spread all over the log
- Updated I-nodes also added to end of the log
- Managing Read Performance
  - in memory index points to latest I-node versions
  - recent log segments are LRU cached in memory

Writes to a Logging File System



Managing Log Recovery Time

- In-memory index is key to performance
  - searching the log is prohibitively expensive
- How do we reconstruct index on restart?
  - rescanning entire log is prohibitively expensive
- Periodically snapshot index to the log
  - update a most-recent-index pointer
- Recovery
  - find and recover last (valid) index snapshot
  - replay all valid log entries after that point

Redirect on Write

- many modern file systems now do this
  - once written, blocks and I-nodes are immutable
  - add new info to the log, and update the index
- the old I-nodes and data remain in the log
  - if we have an old index, we can access them
  - clones, snapshots, time-travel are almost free
- price is management and garbage collection
  - we must inventory and manage old versions
  - we must eventually recycle old log entries

Log (De)Fragmentation

- Eventually the log "wraps-around"
  - unlike circular buffer, not all old entries are stale
  - some old data has not been updated/replaced
  - old log segments still contain (sparse) valid data
- Log defragmentation
  - use index to determine what entries still valid
  - re-copy valid entries to front of log, update index
  - after which old log segment can be recycled
    - same process for NAND flash erasure/wear leveling

Log Defragmentation

Many items near front of log have been replaced, are no longer valid
Copy valid items from (old) start of log to the (new, dense) end of log
The log can now wrap-around and reuse that space at the front.

Reading and Assignments

Reading:

- Reiher: Introduction to Security
- Reiher: Authentication
- Reiher: Access Control

Projects:

- we will help w/project 3B problems in the lab

Supplementary Slides

Causes of File System Data Loss

- OS or computer stops with writes still pending
  - .1-100/year per system
- defects in media render data unreadable
  - .1 – 10/year per system
- operator/system management error
  - .01-.1/year per system
- bugs in file system and system utilities
  - .01-.05/year per system
- catastrophic device failure
  - .001-.01/year per system

Dealing with flaws in the media

- don't use known bad sectors
  - identify all known bad sectors (factory list, testing)
  - assign them to a "never use" list in file system
  - since they aren't free, they won't be used by files
- deal promptly with newly discovered bad blocks
  - Error Correction Coding (ECC) can recover lost data
  - most failures start with repeated "recoverable" errors
  - copy the data to another block ASAP
  - assign new block to file in place of failing block
  - assign failing block to the "never use" list

The ultimate fall-back – Back-ups

- All files should be regularly backed up
- Permits recovery from catastrophic failures
- complete vs. incremental back-ups
- desirable features
  - ability to back-up a running file system
  - ability to restore individual files
  - Ability to back-up w/o human assistance
- must be considered as part of file system design