Operating Systems Principles

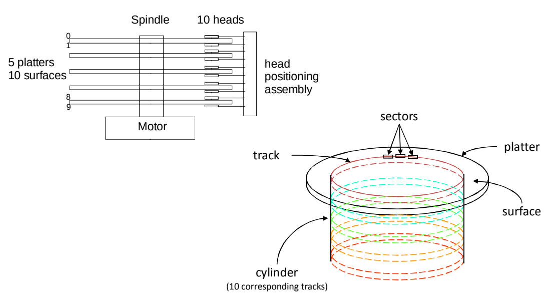
File Systems: Performance & Robustness

Mark Kampe
(markk@cs.ucla.edu)

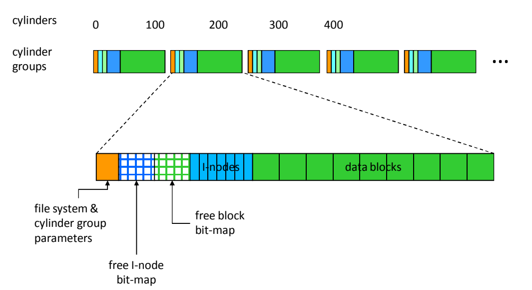
File Systems: Performance & Robustness

11G. File System Performance
11H. File System Robustness
11I. Checksums
11J. Log Structured File Systems

Disk Drives and Geometry



Maximizing Cylinder Locality



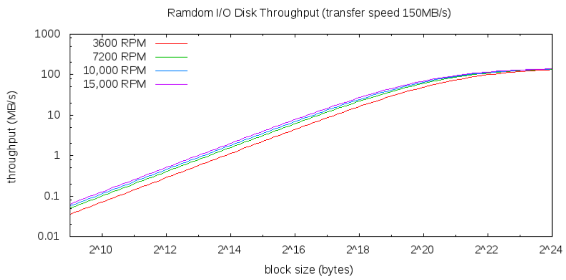
(maximizing cylinder locality)

- seek-time dominates the cost of disk I/O
  - greater than or equal to rotational latency
  - and much harder to optimize by scheduling
- live systems do random access disk I/O
  - directories, I-nodes, programs, data, swap space
  - all of which are spread all across the disk
- but the access is not uniformly random
  - 5% of the files account for 50% of the disk access
  - users often operate in a single directory
- create lots of mini-file systems
  - each with grouped I-nodes, directories, data
  - significantly reduce the mean-seek distance

Seek and Latency Scheduling

- deeper queues mean more efficient I/O
  - elevator scheduling of seeks
  - choose multiple blocks in the same cylinder
  - schedule them in rotational position order
- consecutive block allocation helps
  - more requests can be satisfied in a single rotation
- works whether scheduling is in OS or drive
  - but the drive knows the physical geometry
  - drive can accurately compute seek/rot times

Disk Throughput vs. Block Size



Allocation/Transfer Size

- per operation overheads are high
  - DMA startup, seek, rotation, interrupt service
- larger transfer units more efficient
  - amortize fixed per-op costs over more bytes/op
  - multi-megabyte transfers are very good
- this requires space allocation units
  - allocate space to files in much larger chunks
  - large fixed size chunks -> internal fragmentation
  - therefore we need variable partition allocation

I/O Efficient Disk Allocation

- allocate space in large, contiguous extents
  - few seeks, large DMA transfers
- variable partition disk allocation is difficult
  - many file are allocated for a very long time
  - space utilization tends to be high (60-90%)
  - special fixed-size free-lists don’t work as well
- external fragmentation eventually wins
  - new files get smaller chunks, farther apart
  - file system performance degrades with age

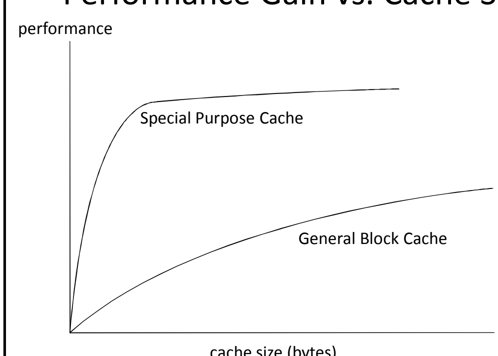
De-Fragmentation

- the fast and easy way … off-line
  - back-up then entire file system to other media
  - reformat the entire file system
  - read the files back in, one-at-a-time
- the slow and hard-way … live, in-place
  - find a heavily fragmented area
  - copy all files in that group elsewhere
  - coalesce the newly freed space
  - copy files back into the defragmented space

Read Caching

- disk I/O takes a very long time
  - deep queues, large transfers improve efficiency
  - they do not make it significantly faster
- we must eliminate much of our disk I/O
  - maintain an in-memory cache
  - depend on locality, reuse of the same blocks
  - read-ahead (more data than requested) into cache
  - check cache before scheduling I/O
- all writes must go through the cache
  - ensure it is up-to-date

Performance Gain vs. Cache Size



Special Purpose Caches

- often block caching makes sense
  - files that are regularly processed
  - indirect blocks that are regularly referenced
- consider I-nodes (32 per 4K block)
  - only recently used I-nodes likely to be re-used
- consider directory entries (256 per 4K block)
  - 1% of entries account for 99% of access
- perhaps we should cache entire paths

Special Caches – doing the math

- consider the hits per byte ratio
  - e.g. 20 hits/4K block (.005 hits/byte)
  - e.g. 10 hits/32 byte dcache entry (.3 hits/byte)
- consider the savings from extra hits
  - e.g. 50 hits/second * 1.5ms/hit = 75ms
- consider the cost of the extra lookups
  - e.g. 1000 lookup/s * 10ns per lookup = 10us
- consider the cost of keeping it up to date
  - e.g. 100 upd/s * 80ns per upd = 8us

When can we out-smart LRU?

- it is hard to guess what programs will need
- sometimes we know what we won't need
  - load module/DLL read into a shared segment
  - an audio/video frame that was just played
  - a file that was just deleted or overwritten
  - a diagnostic log file
- dropping these files from the cache is a win
  - allows a longer life to the data that remains there

Write-Back Cache

- writes go into a write-back cache
  - they will be flushed out to disk later
- aggregate small writes into large writes
  - if application does less than full block writes
- eliminate moot writes
  - if application subsequently rewrites same data
  - if application subsequently deletes the file
- accumulate large batches of writes
  - a deeper queue to enable better disk scheduling

Persistence vs Consistency

- Posix Read-after-Write Consistency
  - any read will see <u>all prior writes</u>
  - 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: 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

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

- queued writes that don't get completed
  - client writes that will not be persisted
  - client 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

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

Practicality – Ordered Writes

- greatly reduced I/O performance
  - eliminates head/disk motion scheduling
  - eliminates accumulation of near-by operations
  - eliminates consolidation of updates to same block
- may not be possible
  - modern disk drives re-order queued requests
- doesn't actually solve the problem
  - does not eliminate incomplete writes
  - it chooses minor problems over major ones

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 - 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 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 <u>idempotent</u>
  - 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

Log Structured File Systems

- the Journal is the file system
  - all I-nodes and data updates written to the log
  - 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

Navigating a logging file system

- I-nodes point at data segments in the log
  - sequential writes may be contiguous in log
  - random updates can be spread all over the log
- Updated I-nodes are added to end of the log
- Index points to latest version of each I-node
  - Index is periodically appended to the log
- Recovery
  - find and recover the latest index
  - replay all log updates since then

Redirect on Write

- many modern file systems now do this
  - once written, blocks and I-nodes are <u>immutable</u>
  - 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 and snapshots are almost free
- price is management and garbage collection
  - we must inventory and manage old versions
  - we must eventually recycle old log entries

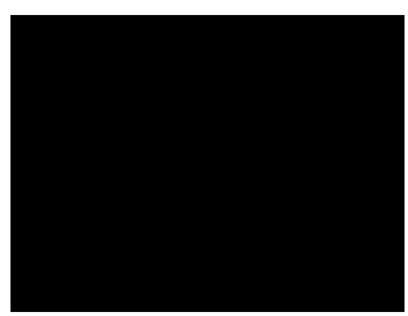
Defragmentation

- a variation of Garbage Collection
  - we may actually know what is unused
  - we are searching for things to relocate and coalesce
- Logging File Systems
  - reclaim (now obsolete) space from back of log
- Flash File Systems
  - create completely free blocks to erase/recycle
- most file systems
  - coalesce contiguous free space for new files
  - recombine fragments created by random updates
  - cluster commonly used files together

Defragmentation Procedure

1. identify stale records that can be recycled
   - versions, reference counts, back-pointers, GC
2. identify next block to be recycled
   - most in need (oldest in log, most degraded data)
   - most profitable (free space ratio, most stable)
3. recopy still valid data to a better location
   - front of the log, contiguous space
   - or perhaps just move it “out of the way”
4. recycle the (now completely empty) block
   - for flash, erase it, add it to the free list

Defragmentation – The Movie



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

Assignments

- for the next lecture:
  - Saltzer 11.1-2 … Intro to Security, Authentication
  - Saltzer 11.6 … Authorization
  - At Rest Encryption

Supplementary Slides

Logical Disk Partitioning

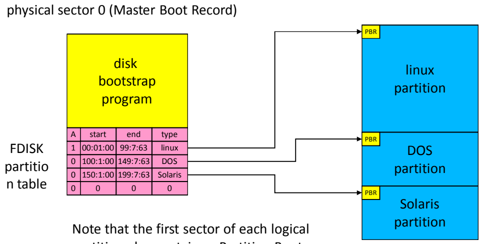
- divide physical disk into multiple logical disks
  - often implemented within disk device drivers
  - rest of system sees them as separate disk drives
- typical motivations
  - permit multiple OS to coexist on a single disk
    - e.g. a notebook that can boot either Windows or Linux
  - fire-walls for installation, back-up and recovery
    - e.g. separate personal files from the installed OS file system
  - fire-walls for free-space
    - running out of space on one file system doesn't affect others

Disk Partitioning Mechanisms

- some are designed for use by a single OS
  - e.g. Unix slices (one file system per slice)
- some are designed to support multiple OS
  - e.g. DOS FDISK partitions, and VM/370 mini-disks
- important features for supporting multiple OS's
  - must be possible to boot from any partition
  - Must be possible to keep OS A out of OS B's partition
- there may be hierarchical partitioning
  - e.g. multiple UNIX slices within an FDISK partition

example: FDISK Disk Partitioning

physical sector 0 (Master Boot Record)



Note that the first sector of each logical partition also contains a Partition Boot Record, which will be used to boot the operating system for that partition.

File System Performance

- we've looked at basic file system operations
  - finding a file based on its name
  - finding a specified block of a file
  - allocating a new block and adding it to a file
- file system data structures should optimize these
  - searches should be short with minimal disk I/O
- we've looked at free list organization
  - try to allocate consecutive blocks to a file
  - minimizes the head motion when we read it back

Compaction and Defragmentation

- file I/O is efficient if file extents are contiguous
  - easy if free space is well distributed in large chunks
- with use the free space becomes fragmented
  - and file I/O involves more head motion
- periodic in-place compaction and defragmentation
  - move the most popular files to the inner-most cylinders
  - copy all files into contiguous extents
  - Leave the free-list with large contiguous extents
- has the potential to significantly speed up file I/O