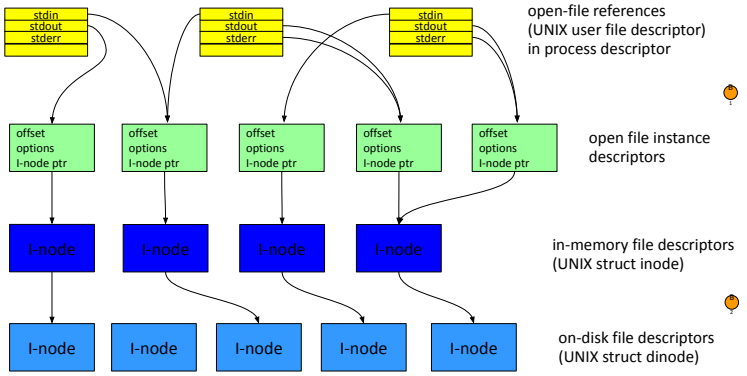
File System Performance

- 11F File System Integration
- 11G File System Performance
- 11K File System De-Fragmentation

Open Files – Levels of Indirection



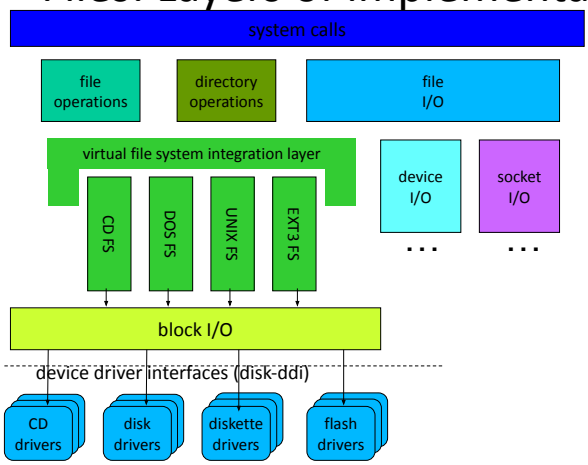
(Open Files – Levels of Indirection)

- open file references (UNIX UFD)
  - array to associate open file index numbers w/files
- open file descriptors (UNIX file structures)
  - describes an open instance (session) of a file
    - current offset, access (read/write), lock status
- in-memory file descriptors (UNIX I-nodes)
  - copy of on-disk file description
- on-disk file descriptors (UNIX dinodes)
  - file description (ownership, protection, etc)
  - location (on disk) of the file's data

File Systems

- file systems implemented on top of block I/O
  - should be independent of underlying devices
- all file systems perform same basic functions
  - map names to files
  - map <file, offset> into <device, block>
  - manage free space and allocate it to files
  - create and destroy files
  - get and set file attributes
  - manipulate the file name space
- different implementations and options

Files: Layers of implementation



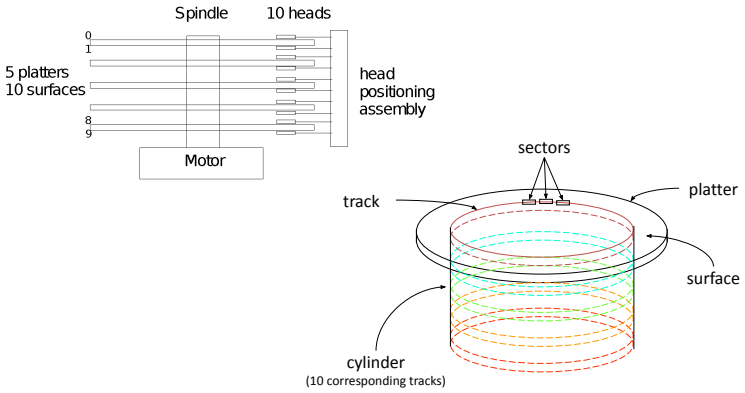
Virtual File System (integration) Layer

- federation layer to generalize file systems
  - permits rest of OS to treat all file systems as the same
  - support dynamic addition of new file systems
- plug-in "bridge" to file system implementations
  - DOS FAT, Unix, EXT3, ISO 9660, NFS, etc.
  - each file system implemented by a plug-in module
  - all implement same basic methods
    - create, delete, open, close, link, unlink,
    - get/put block, get/set attributes, read directory, etc
- implementation is hidden from higher level clients
  - all clients see are the standard methods and properties

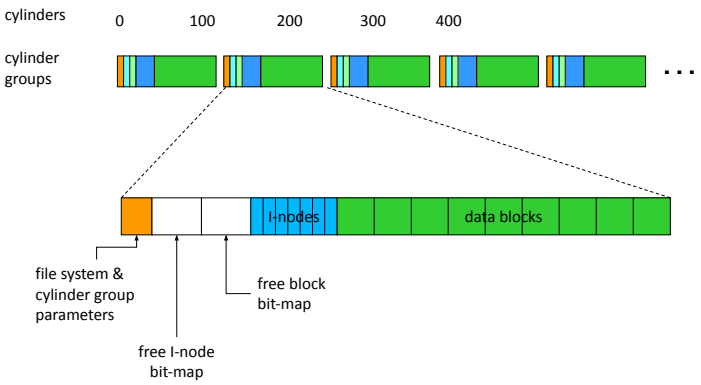
Device Independent Block I/O

- simplifying abstraction – better than generic disks
- an LRU buffer cache for disk data
  - hold frequently used data until it is needed again
  - hold pre-fetched read-ahead data until it is requested
- buffers for data re-blocking
  - adapting file system block size to device block size
  - adapting file system block size to user request sizes
- automatic buffer management
  - allocation, deallocation
  - automatic write-back of changed buffers

Disk Drives and Geometry



Maximizing Cylinder Locality



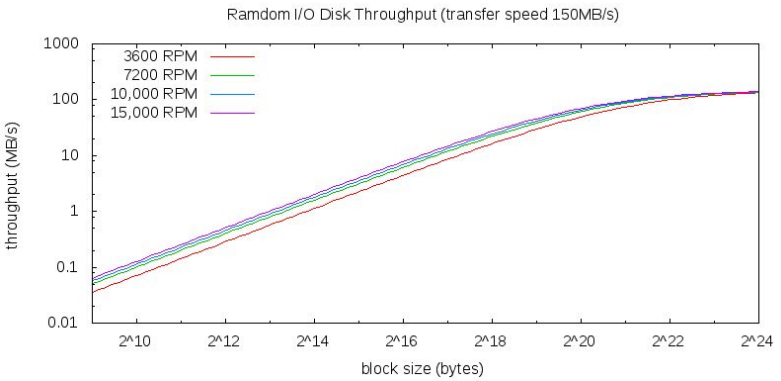
(maximizing cylinder locality)

- 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

Disk Seek/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



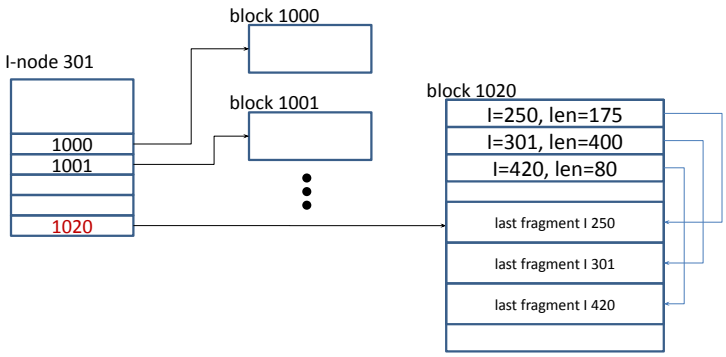
Allocation/Transfer Size

- 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

Block Size vs. Internal Fragmentation

- Large blocks are a performance win
  - fewer next-block lookup operations
  - fewer, larger I/O operations
- Internal fragmentation rises w/block sizes
  - mean loss = block size / (2 * mean file size)
- Can we get the best of both worlds?
  - most blocks are very large
  - the last block is relatively small

Fragment Blocks



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
- recovering performance requires defragmentation

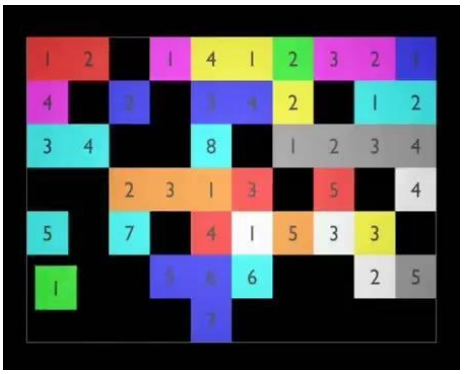
Defragmentation - What

- 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
  - erase/recycle happens in large (e.g. 128K) blocks
- most file systems
  - coalesce contiguous free space for new files
  - recombine fragments created by random updates
  - cluster commonly used files together

Defragmentation - How

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



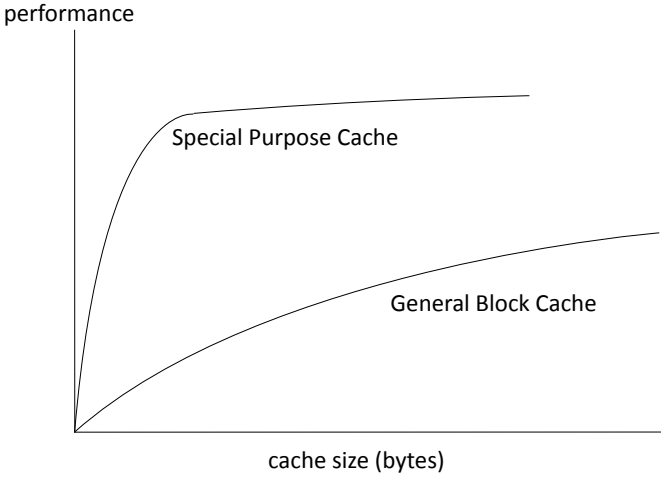
Read Caching

- 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

Read-Ahead

- Request blocks before they are requested
  - store them in cache until later
  - reduces wait time, may improve disk I/O
- When does it make sense?
  - when client specifically requests sequential access
  - when client seems to be reading sequentially
- What are the risks?
  - may waste disk access time reading unwanted blocks
  - may waste buffer space on unneeded blocks
  - a gamble: how often will our guess be correct

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 per second ratio
  - e.g. 2 hits/4K block (.0005 hits/byte)
  - e.g. 1 hits/32 byte dcache entry (.03 hits/byte)
- consider the savings from extra hits
  - e.g. 50 block reads/second * 1.5ms/read = 75ms
- consider the cost of the extra cache lookups
  - e.g. 1000 lookup/s * 50ns per lookup = 50us
- consider the cost of keeping cache up to date
  - e.g. 100 upd/s * 150ns per upd = 15us
- net benefit: 75ms – 65us = 74.935ms/s

When can we out-smart LRU?

- it is hard to guess what programs will need
- sometimes we know what we won't re-read
  - 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 (vs write-thru) 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

Reading and Assignments

Reading:

- Arpaci C42 … crash consistency
- Arpaci C43 … logging file systems
- Arpaci C44 … data integrity
- Arpaci appx I.6-10 … SSD storage management

Projects:

- start looking at project P3B (file system integrity, big)

Supplementary Slides

Block Device Drivers

- generalizing abstraction – make all disks look same
- implement standard operations on their devices
  - asynchronous read (physical block #, buffer, bytecount)
  - asynchronous write (physical block #, buffer, bytecount)
- map logical block numbers to device addresses
  - e.g. logical block number to <cylinder, head, sector>
- encapsulate all the particulars of device support
  - I/O scheduling, initiation, completion, error handlings
  - size and alignment limitations