

I/O Optimization

Steve Martin
Dave Latham

Copyright note: figures and some elements from slides by Hank Levy, CSE451, University of Washington. Thank you, Hank!

Reading

- Peterson and Silbershatz: 8.3-8.7
- Peterson, Silbershatz and Galvin: 9.3-9.10
- Silbershatz and Galvin: 11.1-11.3, 11.5, 12.1-12.6, 13.1-13.3
- Dietel, pp. 305-316, 322-336
- Tannenbaum: 6.1-6.3
- Silbershatz, Galvin, and Gagne: 11, 12

Block size

- Now that you understand how files are stored on a disk, how can we make file operations fast?
 - What's the right block size?
- What if we made all the blocks really small?
 - Advantages:
 - Small I/O buffers
 - Quick transfer times
 - Less internal fragmentation
 - Disadvantages
 - Require more transfers for a fixed amount of data
 - More seeks!
 - High overhead on disk—each block needs to have records.
 - More pointer entries in file descriptors
- Optimal block sizes tend to range from 2K to 8K bytes.
 - This is increasing as technology improves.
- Unix file system uses 4KB blocks.
 - Also uses quarter-block 'fragments' if data doesn't fit into one block.

Disk performance

- Performance depends on a number of steps
 - Seek: moving the disk arm to the correct cylinder
 - Depends on how fast disk arm can move
 - This isn't improving much
 - Rotation: waiting for the sector to rotate under head
 - Depends on rotation rate of disk
 - Rates are increasing, but slowly
 - Transfer: transferring data from surface into disk controller, and from there sending it back to host
 - Depends on density of bytes on disk
 - Increasing, and very quickly
- When the OS uses the disk, it tries to minimize the cost of all of these steps
 - Particularly seeks and rotation

Scheduling disk transfers

- In timesharing systems, there are often many requests to perform disk I/O at the same time.
 - How do we schedule these to maximize performance?
- Disk arm scheduling algorithms
 - FCFS (your old pal)
 - Reasonable when load is low
 - May result in a lot of unnecessary disk arm motion under heavy loads.
 - SSTF (shortest seek time first)
 - Handle nearest request first.
 - Minimize arm movement (seek time), maximize request rate
 - Unfairly favors middle blocks
 - How?
 - Why is this bad?

Scheduling disk transfers (2)

- Disk scheduling algorithms continued . . .
 - SCAN (elevator algorithm)
 - Move arm in one direction, handling all service requests in that direction, then reverse
 - May not get the shortest seek! Skews wait times non-uniformly
 - How?
 - C-SCAN (circular scan, one-way elevator algorithm)
 - Move arm in one direction, handling all service requests in that direction.
 - Then return to request furthest in other direction and repeat.
 - Like a typewriter.
 - Uniform wait times, but higher mean access time than SCAN.
- SSTF has best mean access time
 - But SCAN or C-SCAN can be used if there is danger of starvation.

Rotational scheduling

- Rare to have more than one request outstanding for a given cylinder
- Can schedule based on rotational latency
 - SRLTF (Shortest rotational latency)
 - Schedule requests based on what's going to be under the disk head next.
 - Can be really useful for writing data if we don't have to write back to the same location
 - i.e. long contiguous records.
- Hard to do knowing just the logical block address (LBA)
 - Rotational and seek scheduling can be combined in a useful manner in the onboard disk controller.
 - Need to know angular and radial position of disk head.

Reading blocks

- Once the disk head is in position, now need to read blocks on disk
 - Disk arm movement is not exact science
 - If we're reading in sequentially with no seeks between reads, could run into problems.
 - We could try to read a block just after start of block has passed
 - Solution: allocate file blocks on alternate disk blocks or sectors!
 - Then we haven't passed the block when we want to read it.
 - This is called *skip-sector* or *interleaved* disk allocation.
 - Note that with a semiconductor buffer, this isn't necessary.
- Also takes time to switch between heads on different tracks or cylinders.
 - May want to skip several blocks when moving sequentially between tracks.

File placement

- How do we minimize seek distances?
 - Put commonly used files near the center of the disk
 - Even better if disk use patterns are analyzed and files that are frequently referenced together are placed near each other.
- Frequency of seeks and queue length will be reduced if commonly used files are located on different disks.
 - Why?
 - More on this later (RAID)

Increasing disk performance

- Keep a cache of recently used disk blocks in main memory
 - Recently read blocks retained until replaced
 - Writes go to cache, later written back
 - Typically includes index blocks for open file
 - Why?
- Also use the cache for *read ahead* and *write behind*
 - Load entire track into cache at once!
- Typically works quite well—hit ratios of 70-90%
- Can also do caching in disk controller
 - Most controllers have 64K-4MB of cache/buffer in the controller
 - The Western Digital Special Edition has 8MB!
 - Used mostly as a buffer, not a cache, since main memory is much larger.

Increasing disk performance (2)

- Prefetching and Data Reorganization
 - Disk blocks are often accessed sequentially
 - Can be really helpful to prefetch ahead of current point
 - In order to leverage prefetching, physical layout needs to reflect logical organization of data
 - Logically sequential blocks also physically sequential
 - Since blocks are allocated somewhat randomly, useful to periodically reorganize the disk.
- Data replication
 - Replicate frequently used data at multiple locations on disk.
 - Can make things complicated on writes
 - Why?
- ALIS – Automatical Locality Improving Storage
 - Best results obtained when techniques are combined
 - Reorganize to make sequential, cluster, and replicate.

RAID

- Redundant Arrays of Inexpensive Disks
 - Or "Independent", depending on who you talk to.
- Why have more than one disk in an array?
 - Are reads faster?
 - Are writes faster?
- Observations:
 - Small disks cheaper than large ones
 - Failure rate is constant, independent of disk size
 - Therefore, if we replace a few large disks with lots of small disks, failure rate increases.

RAID (2)

- Having data on multiple disks improves read bandwidth!
- Solution
 - Just could interleave blocks of file across a set of smaller disks
 - RAID 0
 - If one fails, we're in trouble
 - Could mirror all the data
 - RAID 1
 - Wastes space
 - Interleave, and add a parity disk in case of failure
 - RAID 3: bit-interleaved parity
 - RAID 4: block-interleaved parity
 - Can reconstruct data from parity information and remaining disks.
 - Can do parity in two directions for extra reliability

RAID (3)

- Problems
 - If using parity, have to write to parity disk on every write
 - A single disk becomes a bottleneck for the system
 - Solution: interleave parity among the disks
 - RAID 5: block-interleaved distributed parity
- Goals of parallelism in a disk system:
 - Increase throughput of multiple small accesses by load balancing
 - Reduce the response time of large accesses