Max Loh

Lecture Notes: 4/6/09

-Midterm a week from this Wednesday, 7pm, 306 Soda

I/O optimization

-Sun and IBM made huge automated tape libraries, but most optimizations deal with *hard disks.*
-A *sector* is normally 512 bytes, but you can treat it as part of a larger *block.*

Small blocks:

good:

-small I/O buffers (when doing i/o you have buffer usually controlled by operating system. When reading, a block is copied into the buffer and accessed from there. No longer an issue because memory is cheap).

-Quickly transferred: A 512k block would be fast but 16k block would be slow.

-read/write individually has a lot more CPU overhead. seek/latency overhead for every block.

-Less internal fragmentation. (not really an issue these days because memory/disk is so large that loss of a few blocks is less than round-off error) e.g. a 2 tb disk is now 300-400 dollars

bad:

-High overhead on disk: error correction bits, bits between blocks that identify track and cylinder, etc... wasted space. Also, inter-record gaps (space between blocks)

-More entries needed in file descriptor (file maps, tree structure, every block needs a pointer to it).

-Random allocation = more seeks. If they aren't all clustered you'd have to seek a lot.

-Optimum block sizes range from 2K to 8K bytes. Probably bigger these days, because these statistics are old. Not huge, because transfer time hasn't gotten that must faster. perhaps 2k - 16k bytes now.

-Bekreley Unix uses 8K blocks basic (hardware) block size in VAX is 512 bytes.

-Berkeley Unix also uses fragments that are 1/4 the size of the logical block size. Instead of just allocating blocks, you can also use "quarter blocks." If a block isn't used it can release a block in quarter-block increments. This allows you to allocate large blocks but not waste space.

Disk Arm Scheduling:

-Suppose you have a queue of requests to the disk scattered around disk surface (see figure 1.1).

Max Loh

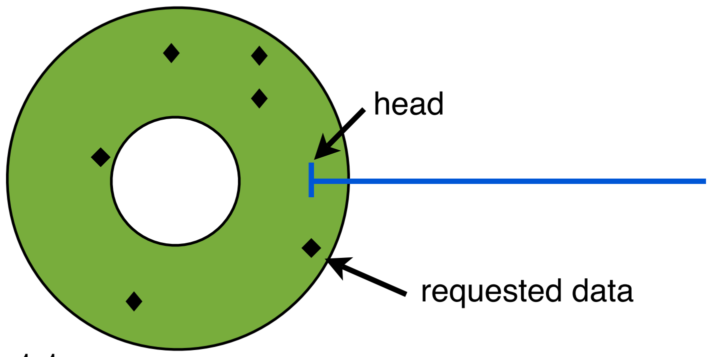

Fig. 1.1

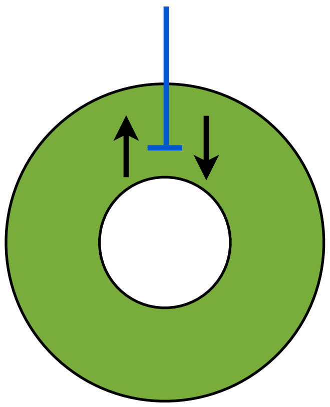

Fig. 1.2 (scan): head seeks upwards, then downwards, like an elevator.

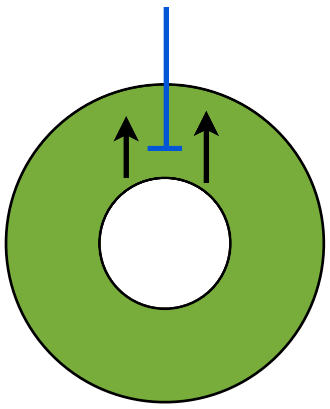

Fig. 1.3 (Cscan): head seeks in one direction only.

Algorithms:

FIFO - first come first serve (self-explanatory)

SSTF - shortest seek time first: handle nearest request first. Reduces arm movement, results in greater overall disk efficiency.

-Problem: starvation, e.g. disk heavily loaded, 3 open files. Two files near center of disk, other near edge. Disk ignores last file.

Scan - like an elevator, up and down. Move arm in one direction, servicing requests, until there are no additional requests in that direction. Then, reverse direction and continue (see figure 1.2)

-Problem: files near the center get visited twice as much as files located near the edges.

-Works well under heavy load but may not get shortest seek time.

-Neglects files in periphery (edge) of disk

CScan - (circular scan) like a one-way elevator. Moves only in one direction. When it finds no further requests in the scan direction it returns immediately to the furthest request in the other direction, and it resumes the scan (see fig. 1.3).

-This treats all files equally, but somewhat higher mean access time than Scan.

-Someone did an experiment and assumed requests were scattered all randomly around the disk surface and independent, and simulated this. They found that FIFO is terrible, the other three same but SSTF a little better. Performance estimate: FIFO 1, sstf 4.5, scan 4, cscan 4.

-What's wrong with the studies? Accesses are more structured, not randomly scattered. We should expect a lot of spatial locality. One 10 MB file allocated randomly in 4k blocks is dumb. What we should do is lay out 2500 consecutive blocks, and read sequentially. Also, we usually don't have hundreds of files open (3 or 4 at most). If we just read a block, chances are the next block is the one we want next.

Question: Has anyone ever done a smarter/adjusted version of the study?
Answer: Some people traced the performances of the algorithms on a real system. Similar results.

Another problem: If you have 3-4 files and 2 processes, a queue of 50 disk requests is unlikely (more like 2 or 3 disk requests at a time). Most of the time, disk queue is low.

However, imagine a database system doing query processing for a bank, 25 tellers all doing queries. Accesses are really all random. It's modern so you have a 2TB disk. In that case you really would have this circumstance and it would make sense to make a better algorithm than SSTF (Traveling salesman problem).

The distance would not be Euclidean (see picture). You'd have to take into account rotational latency and head seek time. Unfortunately, only disk controller knows location of head and everything, and it doesn't have information about pending requests.

In conclusion, SSTF has best mean access time. Scan or CScan can be used if there is danger of starvation, but SSTF is mostly preferable. Note that in many circumstances, the disk arm scheduling algorithm has little effect on performance, especially if seeks are seldom required (as in whenever there is contiguous allocation of data on hard disk).

Of or related to real-world business (professor going off on tangent, probably not on exam):

-Problem with disk companies: Comprised of physicists and mechanical engineers. Mechanical engineers know about spinning heads. physicists know about magnetism. no one knows about computer/operating systems. All they know is the signals coming over that wire. They turn their product 4 times a year, always their best technology, but no understanding of systems, can't think higher-level of optimization in context of a system.

-Disk companies suffer from *Oligopsony* -- economy dictated by a small number of buyers. Make a popular disk, make a small amount of money. Make an unpopular product, go bankrupt. I/O business is sketchy.

Rotational Scheduling

-Most of the time, no one bothers. It is rare to have more than one request outstanding for a given cylinder.

-SRLTF (shortest rotational latency first) works well.

-Useful for writing data, if we don't have to write back to the same location. Let's suppose you read a bunch of blocks on disk and want to rewrite them. Two choices: write them back to where they were, or write somewhere else and change File Descriptors. In the latter case, we can choose to write directly under the head as disk spins. No rational latency.

-Rotational scheduling is difficult using logical block address (LBA): modern disks, at the level of the connector, do not know the arm or rotational position or block number. So you can guess, but that's about all you know.

-However, rotational and seek scheduling can be usefully combined (shortest time to next block) if done inside disk controller.

Skip-sector or Interleaved disk allocation

-Imagine you are reading the blocks of a file sequentially and quickly, and file is allocated sequentially/contiguously.

-Problem: usually, you will tend to read a block just after start of the next block has already passed (see figure 2.1).

-Solution is to allocate file block to *alternate* disk blocks or sectors. That way the block won't have passed before we want to read it. See figure 2.2.

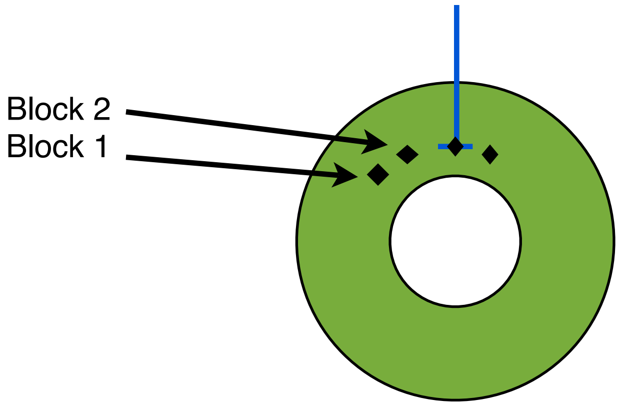

Figure 2.1: The head reads block one but an interrupt prevents it from reading block 2. By the time the interrupt completes, the disk has spun past block 2. Now we must wait for an entire spin cycle.

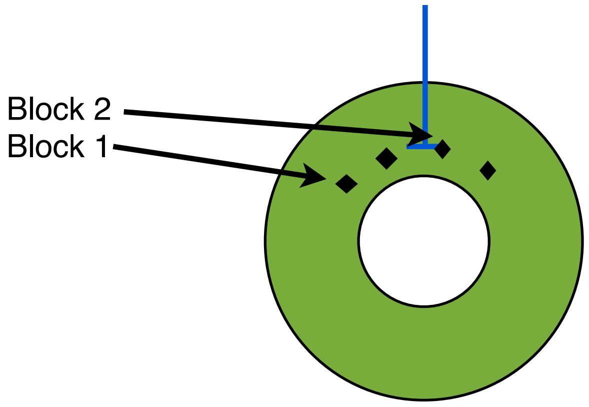

Figure 2.2: If we alternate the numbers of the blocks, it will accommodate for interrupt disruptions while reading the disk.

Track offset for head and cylinder switching

-It takes time to switch between heads on different tracks or cylinders. Thus we may want to skip several blocks when moving sequentially between tracks, to allow the head to be selected. See figure 3.

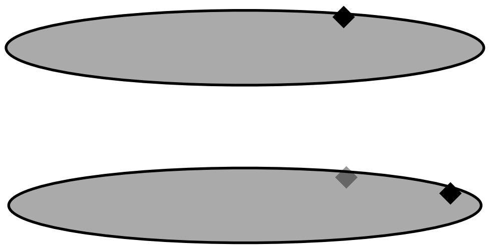

Figure 3: Switching from the top disk to the bottom disk in illustration takes time. Instead of having the next block continue at the same location we left off (illustrated by gray dot), skip some blocks to allow for switching latency (illustrated by black dot). This avoids a situation in which we “miss” the next block and the head must wait an entire disk rotation before continuing.

File Placement

-Seek distances will be minimized if commonly used files are located near the “center” of disk. See figure 4.

-Even better results if reference patterns are analyzed, and files that are frequently referenced together are placed near each other.

-Frequency of seeks and queueing for disks will be reduced if commonly used files (or files used concurrently) are located on different disks. E.g. spreading the paging data sets and operating systems data sets over several disks.

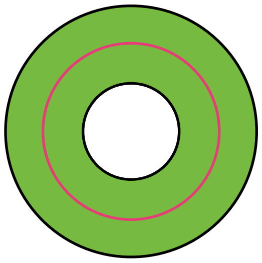
Figure 4 (the red circle illustrates the “center” of the disk)

Q: disks with multiple heads?
A: cylinders have one head per surface. There are disks with multiple heads running in parallel, reading 8 tracks at once, 8x bandwidth. I don’t think anyone has had a disk where you have multiple arms serving the same set of platters. If so, made long time ago. Did have one case: One arm serving multiple sets of platters. See fig 5. The benefit of this model is that it is cheap (heads are expensive, platters are cheap).

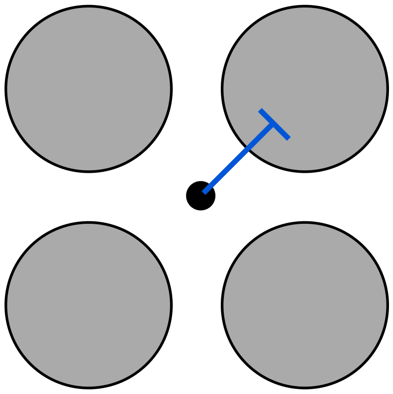
Figure 5: one head serving four platters.

Disk Caching:

-By electronic standards, disks are slow (disk access takes milliseconds, memory access takes nanoseconds).
-We would like to keep a cache of recently used disk blocks in main memory. When reading blocks, store them in the cache. When writing blocks, write to the cache
-Recently read blocks are retained in cache until replaced.
-Writes go to disk cache, and are later written back.
-This scheme typically includes index blocks for an open file.

-Problem: OS always makes the assumption that when writing to disk, it's stored and safe. If it thinks it's on disk but really on cache/memory, then when you lose power you have a problem.

-Solution 1: power backup.

-Solution 2: don't complete I/O until it is really on magnetic physical storage.

-Solution 3 (really truly paranoid): write it to disk, read it back to compare if it got written correctly. High-security bank records do this. There is a trade-off with performance, of course.

-Caches are used for read-ahead and write-behind. Just put the data into cache, and cache writes to hard disk as fast as it can.

-Disk caches work well with hit ratios of 70-90%

-It is possible to cache in the disk controller itself, instead of main memory. Most controllers these days 512k to 16MB of cache/buffer in the controller. Mostly useful as buffer, not cache, since the main memory cache is so much larger.

Prefetching and Data reorganization:

-A lot of files are read sequentially. To reduce seeks, we should therefore allocate the files sequentially. If blocks are laid out sequentially, we can just keep reading the disk quickly and continuously.

-One approach: If the data is too fragmented, dump the whole disk and write it back sequentially. However it can break, lose data.

-It is useful to make sure that the physical layout of the data reflects the logical organization of the data -- i.e. logically sequential blocks are also physically sequential. Thus it is useful to periodically reorganize the data of the disk.

Data replication:

-Frequently used data can be duplicated so you can just seek to the nearest one and read it.

-This means that on writes, extra copies must either be updated or invalidated.

ALIS - automatic locality improving storage:

-Combines many techniques: writing data sequentially, data replication, etc.

-*Autonomic computing*: computer system manages itself. Smart storage system.

-Best results obtained when techniques are combined: reorganize to make sequential, cluster, and replicate. Must make sure it's reliable. If system messes up, you will lose a large amount of data.

RAID

Observations:

-Small disks cheaper than large ones (due to economies of scale)

-1980's small disks cost millions, large disks cost thousands. Most people just bought numerous small disks.

-However, failure rate is constant, independent of disk size. Mean time is 50 years for 1 disk. But 50 disks, 1 per year approximately.

Solution:

-Therefore, let's design a system where there's redundancy so if disk fails we can reconstruct data.

-Interleave the blocks of the file across a set of smaller disks, and add a parity disk. See fig 6: Mirror disks. Parity disk is the XOR of all the bits on the other disks.

-We presume only one disk failure, and we know which disk failed; therefore, we can reconstruct the entire failed disk.

-Advantage: Improves read bandwidth

-Problem: This means that we have to write the parity disk on every write. This becomes a bottleneck.

-Solution: Interleave on a different basis than the number of disks. That means that the parity disk varies, and the bottleneck is spread around.

-Types of RAID:

RAID 0 - ordinary disks
RAID 1- replication (figure 6.1)
RAID 4 - parity disk in fixed location (figure 6.2)
RAID 5 - parity disk in varying location (figure 6.3)

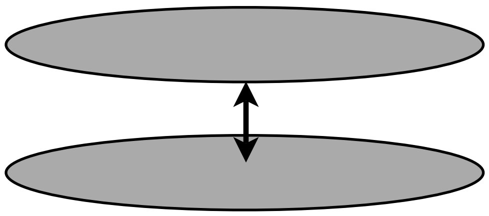

Figure 6.1: simply have a replica disk to supplement the original.

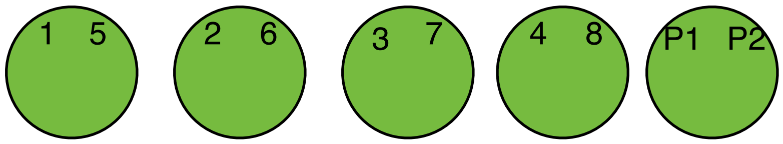

Figure 6.2: Parity disk keeps XOR of interleaved disks

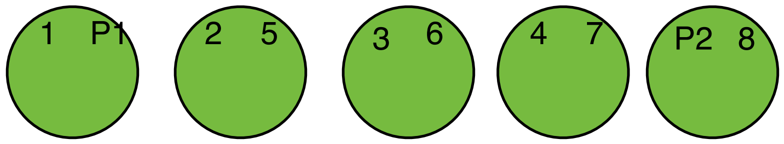

Figure 6.3: Parity disk in varying location

"Berkeley invention" Log-structured file system:

-Suppose you are doing a lot of disk caching, and it is working pretty well. Not very many disk reads, because most reads can be read from the data cache.

-However, there is a reliability issue: Every write is going to the surface of the disk. So now, all disk traffic is writes. Every write typically requires a seek.

-A *log-structured file system* writes all the blocks sequentially, and updates file descriptors to point to the correct place. It is called "log" because it writes in the shape of a log. This technique, although successfully implemented in some research projects, is typically superfluous.