CMPSCI 377 Operating Systems **Fall 2012**

Lecture 18: November 13

Lecturer: Prashant Shenoy *TA: Sean Barker & Demetre Lavigne*

18.1 Secondary Storage

To read from a storage device requires physical movements: the disk head must be moved to the right track, and the platter must be rotated so that the proper sector is underneath the disk head. Thus to minimize the time to make a disk access, it is necessary to minimize both the seek time and the rotational latency. Rotational latency is governed by the rate at which the disk spins (measured in revolutions per minute, or RPMs). Simple solutions like making the disk smaller or spin faster are generally not feasible due to the physical constraints of building a reliable disk device that has sufficient storage space. Instead, the OS and the disk driver must find a way to either schedule disk operations in order to minimize disk head movement (e.g. grouping together reads or writes to the same track), or laying out data on disk in such a way that accesses tend to be made to the same portion of the disk. One approach is to place commonly used data on the outer or innermost tracks. Another challenge is determining the optimal sector size. Since it is possible that a seek will be needed between every disk access, having a large sector size can lead to more data being read between each seek operation. However, if it is too large, then internal fragmentation can become an issue since the sector is the smallest unit within the disk.

Some of these challenges may be alleviated by the advent of solid state flash drives. SSDs do not require mechanical moving parts, so there is no seek delay or rotational latency.

18.2 Disk Head Scheduling

Disk Head Scheduling algorithms are used by the operating system to try to minimize the time wasted due to unnecessary seeks or rotational latencies. The basic ideas is that since at any time there may be multiple disk requests waiting to be serviced, the OS can attempt to reorder those requests in order to reduce the length and number of seeks required.

The **First Come First Served** (FCFS) algorithm simply services requests in the order that they arrive without any reordering. This is the simplest scheduling algorithm, and its performance can be very variable depending on the disk access pattern. If requests happen to arrive in sequential order or if the system is only lightly loaded, then FCFS can perform well. However, if reads are not well ordered, performance can be very bad.

Example Requests: (head starts at 30), 65, 40, 18, 78
Order of Seeks: 65, 40, 18 78
Distance of Seeks: $|30 - 65| + |65 - 40| + |40 - 18| + |18 - 78| = 35 + 25 + 22 + 60 = 142$

Shortest Seek Time First (SSTF) attempts to optimize the requests by reordering requests to minimize the seek time between each request. This can improve performance by minimizing each seek time, but it does not necessarily lead to an optimal overall cost. The SSTF algorithm can be implemented using a doubly linked list of requests which is sorted by seek time. However, like other greedy algorithms we have seen (e.g. Shortest Job First), it has the potential of starving requests. It also tends to favor the middle third of the disk causing the outer, or innermost tracks to have high average seek times. Running requests with the shortest seek times can also cause the rotational time to increase (which will degrade performance). There is a more complex algorithm, Shortest Access Time First (SATF), which computes the seek + rotation time of each request and runs the minimum cost request first. SATF is not done in hardware and it can only compute an approximation of the actual solution. This is because SATF is computing a result that is similar to the Travelling Salesman Problem (TSP) which is NP-hard.

Example Requests: (head starts at 30), 65, 40, 18, 78
Order of Seeks: 40, 18, 65, 78
Distance of Seeks: $10 + 22 + 47 + 13 = 92$

The **SCAN** disk head scheduling algorithm tries to alleviate some of the problems of SSTF by simply moving the disk head back and forth across the disk, servicing requests as it passes over them. For example, if there are 100 tracks on the disk it will travel from 0 to 100, then back from 100 to 0. This prevents the disk head from "bouncing back and forth" within a portion of the disk, as is possible with SSTF. A trivial optimization to SCAN is to make the disk head only scan as far as the last request in the current direction of travel. The SCAN algorithm requires only a sorted list of requests in address order.

Example Requests: (head starts at 30), 65, 40, 18, 78
Order of Seeks: (assume head is traveling to lower numbers first) 18, 40, 65, 78
Distance of Seeks: $12 + 22 + 25 + 13 = 72$

A variant of SCAN is called **Circular SCAN** or C-SCAN. This is a modification of SCAN that attempts to make wait times more consistent across requests. Note that for the normal SCAN algorithm, it is likely that once the disk head reaches the end of a disk, most of the remaining requests to be serviced will be at the other end of the disk, causing them to have a high wait time as the disk must travel back to them. C-SCAN treats the disk as a circular buffer, and thus services requests 0 to 100, then immediately loops back to service 0 to 100 again.

Example Requests: (head starts at 30), 65, 40, 18, 78
Order of Seeks: 40, 65, 78, 18
Distance of Seeks: $10 + 25 + 13 + 60 = 108$

18.3 Improving Disk Performance

It seems reasonable to allocate disk blocks in contiguous chunks as requested by the OS. Keep in mind however, that disks are always rotating, and thus if the disk may spin past an adjacent sector in a contiguous region before the request for that sector arrives. As a result, **disk interleaving** can be used to allocate blocks that are not physically contiguous, but that are "temporally contiguous" based on the rotational speed of the disk. This means that if the disk will have spun past three blocks during the time it takes for a second disk request to arrive, that blocks should be allocated with that level of separation. As a result, the disk blocks used for several different files may be interleaved in a region of the disk rather than each being given its own contiguous set of blocks.

A second performance optimization is **read ahead**. The goal of read ahead is to reduce the number of seeks needed by trying to predict which blocks a user will request and reading them into a cache ahead of time. If the disk controller guesses correctly and a request is made for one of the blocks stored in the buffer, then it can be instantly serviced without requiring a full disk access. Note that this can be done efficiently since the disk head can read in blocks as the platter rotates to service the next request. However, to be effective, the system needs an effective way of predicting which blocks should be read, particularly since the cache in the disk controller will be orders of magnitude smaller than the full disk. We saw earlier that pre-fetching virtual memory pages was difficult due to the difficulty in predicting which pages will be used next. Fortunately, with a disk it can be easier to make these predictions since many disk requests tend to access sequential blocks within a file. Read ahead is used by both the disk itself and the operating system. The disk will read the next sequential blocks on disk into its buffer in case they will be requested soon in the future. The OS will request the next part of a file for a program before program even asks for it. This optimization removes the need for disk interleaving because adjacent blocks will simply be read into the buffer and not skipped over.

18.4 Solid State Drives

Solid state drives, or SSDs, are a relatively recent technology that makes most of the previous discussion about optimizations obsolete. SSDs are random access devices with no moving parts that use flash storage (which is basically non-volatile RAM). Each block of data on a SSD can be accessed directly by its block number without any seek or rotation times. This causes reads of data to be very fast but writes are slower. Writing data to an SSD is slower because the technology requires a erase cycle before new data can be written to a block. There is also a limit on the number of times that each block can be written to. Wear-levelling tries to write to blocks randomly to keep each block at about the same "age". For this reason, blocks are not immediately overwritten when the data in that block changes. The new data is written to a clean block and the old block is scheduled to be garbage collected later. Even with this caveat, both reading and writing to SSDs are much faster than a traditional spinning platter disk. They are also much more energy efficient than traditional disks.

18.5 Tertiary Storage

In the storage hierarchy there are registers, caches, main memory, secondary storage (e.g. hard disks), and tertiary storage. Tertiary storage devices are traditionally larger and cheaper than disks, but also slower. They are mainly used for archival data storage—for example tape drives or optical storage like CDs and DVDs. This has been changing, however, with the low cost of basic hard disks.

18.6 RAID Storage

RAID (redundant array of independent disks, or redundant array of inexpensive disks) uses multiple disk drives to provide better reliability by creating redundancy. The basic idea is to use many disks and make them "look" like a single logical disk to the OS or user. Performance is also increased by being able to parallelize reads (and potentially writes depending on the RAID level). There are many *levels* of RAID but most are combinations of the first six levels. The levels are as follows:

1. RAID 0: non-redundant striping. This level of RAID provides no fault tolerance but offers increased performance and a larger logical pool of storage space.
2. RAID 1: mirrored disks. Each disk has a full copy of itself on another disk.
3. RAID 2: memory-style error-correcting codes.
4. RAID 3: bit-interleaved parity. Data is spread out among multiple disks and parity information is kept on its own separate disk.
5. RAID 4: block-interleaved parity. Data on multiple disks and a dedicated parity disk.

6. RAID 5: block-interleaved distributed parity. Data and parity spread out over multiple disks. The parity information doesn't have its own dedicated disk but rather is kept on the data disks as well.
7. RAID 6: P + Q redundancy. This is much like RAID 5 except that there is extra parity information so that two disks (instead of one) can fail without loss of data.

Parity information is calculated using the XOR operation. XOR is a binary operator that returns true when only one of the two inputs are true and returns false otherwise. For example, if we have some data A and other data B we can compute parity P by: A XOR $B = P$. Now if we lose A, then it can be recovered by: $A = B$ XOR P.