4/6/09

I/O Optimization

- Midterm a week from wednesday
- Tape is pretty much used for archival backup these days. Rarely do you find people who try to optimize tapes. In high end computer systems there's tape libraries.

- Block Size Optimization
    - Small Blocks
        - Small I/O buffers
            - Discuss I/O buffers - used for reads and writes.
        - Are quickly transferred
        - Require lots more transfers for a fixed amount of data.
        - High overhead on disk - wasted bytes for every disk block. (Inter record gaps, header bytes, ERC bytes).
        - More entries in file descriptor to point to blocks (Inode)
        - Less internal fragmentation
        - If random allocation, more seeks.
    - Optimal block sizes tend to range from 2K to 8K bytes
        - Optimum increasing with improvements in technology
    - Berkeley Unix uses 4K blocks. (now 8K?) 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.

- Small block sizes
    - In days when memory was tiny you cared about buffer size. Interesting 30-35 years ago, but not an issue now because memory is cheap.
    - If you have small blocks and you read and write them individually, it's a lot more transfers.
    - Every transfer has overhead
    - Every physical block has overhead on the disk. Error correction bits, identifying bits, inter record gaps
    - Less internal fragmentation. Not really a problem these days because disk and memory are so large these days.
    - People have done some analysis of optimal block sizes. Probably bigger these days. Not sure what exact number is. Not going to be huge, because I/O devices haven't gotten faster by much. Maybe instead of 2K to 8K, 2K to 16K depending on parameters.
    - Berkeley Unix doesn't just allocate blocks, but allocates quarter blocks.
        - When you're writing a file, it can release blocks you didn't use in quarter block increments.
        - You can allocate a lot of blocks but not waste a lot due to fragmentation.
- Disk arm scheduling
    - Suppose you have a queue of requests to the disk, and they're scattered around on the surface of the disk. What order do you do this in?
    - Optimize seeks. Don't worry about rotational latency.
    - A few options
        - FIFO. We can do much much better than this.
        - SSTF (Shortest seek time first). At any point go to the closest one.
            - Problem with this approach is that you tend to get starvation.
        - SCAN. Basically like an elevator. Goes back and forth in direction and services until the end, then goes the other way.

        - If you think about it for a while, you'll realize there's also a bit of a starvation problem here. Requests in the middle of the disk are visited twice as often as the requests at the ends of the disk. Unfair algorithm.
    - CScan. Only goes in one direction. Does a quick return, then services requests again in the same direction. This means that all regions of the disk get equal service.
  - Research found that FIFO was really terrible. The others were relatively equal, with SSTF being a little better than the other two.
    - The problem with this study was that the accesses are not randomly scattered throughout disk. There's usually spatial locality.
    - If you have data laid out sequentially, the probability of a seek isn't very high, since it's likely the next block will be part of the same request.
    - If you have two files, you're not going to have a lot of outstanding requests. Usually you're going to have a couple. Disk utilization is something like 5%. The other 95% the disk is idle. The scheduling algorithm doesn't really make a lot of difference.
    - In other systems this would matter. Imagine a database system in a bank. Accesses are random. Would really make sense in this situation to have a better algorithm.
  - This is the traveling salesman problem. N-requests pending, and you need to visit them all. However, the number of cities keeps changing, so you might need to keep calculating your route. It's also not euclidean.
  - If you look at the world of I/O devices. Most of the time you lose money, but a few are profitable.
  - The major thing is that most of the time the queue is short and the files are laid out sequentially, so there isn't much disk optimization to be done.
- Rotational scheduling
- Skip-Sector of Interleaved disk allocation
  - Average rotational latency was very close to an entire rotation.
  - To overcome this we can number the blocks alternately, so our next I/O get's there before we rotate to the next block on disk. If it's still not enough time, you can number the blocks every third block.
  - This depends on how your I/O commands work. If you read many blocks at a time, you'll want them sequential. If it's reading a block at a time, you might want to use interleaving.
- Track offset and cylinder switching.
  - It takes time to switch between heads. You have amplifiers and switching transients. It may take several milliseconds to switch between heads.
- File placement
  - If you put your most popular files near the center of the disk, you will have shorter seeks.
  - The unpopular files are placed on the outer edges of the disk where the seeks are longer.
  - If you have two disks, and two files that are typically used at the same time, it's a good idea to put them on separate disks.
- Caching
  - By electronic standards, disks are slow. We're talking milliseconds to read and write, while we need only nano seconds to read and write memory.
  - When we read blocks, we keep them in the cache. When we write blocks, we can write to the cache, but there's a catch. The catch is that the operating system makes the following assumption: When we write to disk, it's written to magnetic storage and it's safe. If we think it's on

disk, and it's really on semiconductor storage that loses its contents when it loses power, you have a problem.
    - The so-called cache in the drive isn't really a cache, but is simply a buffer.
- Data replication
    - If there's certain things on disk that are read a lot, we can have multiple copies. We seek to whichever copy is closest. The catch is that what you're replicating is writeable, writing to a copy forces you to update all the copies.
- RAID
    - Small disks became enormously cheaper per byte.
    - The obvious thing to do is to buy a lot of small disks rather than a big disk.
    - The failure rate of small disks wasn't great.
    - The idea is to design a system where there's redundancy. If a disk fails, we don't lose data because we can restore it.
    - The parity disk is the XOR of all the corresponding bits in the other disks.
    - Parity in memory allows you to detect errors. Parity in memory allows you to repair the error.
    - RAID 4: The bottleneck is that every time we write any of the data disks, we need to rewrite the parity block. Your write bandwidth is cut by a factor of 4, since every write writes to parity disk. The solution to this, is to split the parity between all the other disks.
    - Disks fail in two ways. One is randomly, and the other is part of a conspiracy.
- Log Structured File System
    - Supposed you're doing a lot of disk caching. It works well, so you're probably not doing many reads. If you want reliability, you're still doing a lot of writes.
    - Now, all traffic is writes, so bottle neck is writes.
    - Writes implies seek. Why don't we write stuff sequentially?