A Comparison of Disk Scheduling Algorithms

COSC 2P13 – Assignment 2

Matt Laidman
5199807, ml12ef

February 27, 2015

Table of Contents

Disk Scheduling Policies

Since the first modern computers; there has been a significant difference in the speed of processors, the speed of main memory, and main memory access time compared to the speed of hard disks, and hard disk access time. Today, the fastest hard drives have an average seek time of two milliseconds, while main memory typically has an average access time of 100 nanoseconds.

Data is stored on tracks on the disk that are separated into sectors. When a hard disk receives a data request, the disk head must first move horizontally to the appropriate track. The head must then wait for the disk to spin until the beginning of the sector that the data is on is underneath the head, at which point the head will read the data. The less the head has to move between I/O requests, the faster the access time will be.

This gap is most likely not going to shrink as long as hard disks are still used as secondary storage. It therefore necessitates that the order in which data requests to the disk are sent in such a manner that the disk head should travel the least amount of distance to get to the physical location on the disk. However, it is also worth considering that the disk should be fair and not make one application wait longer than it reasonably should.

These requirements have given birth to a number disk scheduling algorithms, each of which is fairly similar to each other. Ten of these algorithms will be discussed. These are:

- FCFS
- PRI
- SSTF
- SCAN
- C-SCAN
- N-Step SCAN and FSCAN
- Eschenbach Scheme
- LOOK and C-LOOK

These algorithms create and order a queue for the data requests in order to ensure fair, optimal disk access time. Once a request has been completed by the disk, the algorithm sends the next request in the queue.

FCFS

The most obvious queue type; first come first serve, or first in first out, only takes into consideration the order in which requests are sent. This algorithm does not take the physical location of the data on the disk into account at all. As such, the disk head may have to travel extreme distances, bypassing data that it could have accessed along the way.

For example; if data was requested from the most inner track, followed by data requested from the most outer track, followed by data requested from a track somewhere in the middle, then the head would have to skip over the track in the middle to service the second request after the first request was completed.

This could potentially create large gaps of wasted time where a user or application is stuck waiting for data. The order of the queue is entirely resultant on the order that requests were sent. If they are not sent in the optimal order, then time is being potentially wasted.

There is also the possibility to use a last-in, first-out queue implementation. This would however likely be only used in special circumstances and on specialty operating systems.

PRI

A priority based policy is one that is handled by the CPU, and is outside the control of the disk management software. The I/O queue is ordered based on an applications priority level. This method of disk scheduling is not intended to optimize disk access time, rather it is used to allow an operating system to ensure certain jobs marked with a higher priority will be served first by the disk than those with lower priority.

One use for this is to give short I/O procedures a higher priority than those that are longer, as the short ones will complete quicker. This however leaves the problem of long jobs potentially having to wait an excessive amount of time.

SSTF

Shortest seek time first is an algorithm that places precedence on the physical location of the requested data rather than the time at which it was sent. When a request is completed, the next request in the queue with the physical location relatively closest to the drive head is sent to the disk.

This method ensures that the drive head will not waste time moving to a track that is far away, when there is already a request for data on a track that is close. This method could

however lead to the problem of “arm stickiness”, where in a series of data requests sent in a short time period have a physical location on the same track. This algorithm will send those requests first causing the arms relative position to get “stuck” on that track. Data requests that are far away from the head will be given a low priority in the queue and as such, the request will take longer to complete.

SCAN

The SCAN disk scheduling algorithm is also known as the elevator algorithm because it causes the disk head to behave similar to the way an elevator does. It is an algorithm designed to be an alternative to FCFS, however one that does not have the potential for a request to be stuck in the queue until it is emptied.

With a PRI or SSTF queue, jobs with a low priority or jobs that are on a far track could potentially not be serviced until all other jobs in the queue have completed if the other jobs get places higher in the queue.

With the SCAN algorithm; the disk head is required to continue servicing sequential jobs in its current direction until it reaches the inner or outer most track, at which point it will stop and process jobs in the opposite direction, in order.

The SCAN policy as such behaves similarly to the SSTF policy, in the sense that all jobs that are in the current direction of the head are queued sequentially by closest physical location. This method also has the potential for arm stickiness.

C-SCAN

The C-SCAN, or circular SCAN, policy behaves similar to the SCAN policy, however the disk head is allowed only to move in one direction. When the head reaches either the inner or outer most track (depending on the direction the head is restricted to) it moves all the way back across the disk to the opposite side and starts reading again in the same direction.

This algorithm allows for a more fair I/O wait time for users and applications requesting data from the disk, as the head will be moving in a more evenly distributed pattern. If a request's data resides on the track opposite the track the head is moving towards, rather than services all requests in the middle of the disk before the far one, the head will reset itself to the far track and service it first.

N-Step-SCAN and FSCAN

N-Step-SCAN and FSCAN are two disk scheduling algorithms that are very similar. They both aim to minimize the effect of the “arm stickiness” caused by the head not moving when concurrent data requests occur for the current track.

With N-Step-SCAN, the request queue is broken into *N* sub queues which are then processed sequentially. When one queue is being processed and emptied, no request can be added to it. Each queue is processed using the SCAN algorithm. With this policy there is no way of knowing then the queue a data request is placed in will be serviced.

The size of *N* is a fixed integer, however it's value is important. For very heavy loads; a large value of *N* is optimal. However for smaller loads, a smaller value of *N* is optimal. If *N* is too large however, it starts to behave similar to a SCAN algorithm.

With FSCAN, the request queue is broken into two sub queues that both use the SCAN algorithm. Just like N-Step-SCAN, when one queue is being serviced, all other requests are placed into the other. With this implementation however, the servicing of relatively newer requests occurs after the servicing of relatively older requests.

Eschenbach Scheme

Eschenbach Scheme takes a different approach than the other scheduling algorithms and attempts to optimize rotational delay – the time the head has to wait for the disk to spin until the desired sector is underneath it - while optimizing seek time.

With Eschenbach Scheme; the disk head moves in a circular fashion similar to C-SCAN, however the head will read in one entire track at a time servicing all requests that are on that track for one full rotation of the disk. If two requests require the same sector to be read, then the second request must wait for the head to pass by that track again.

This algorithm is most efficient when servicing heavy I/O request loads. When frequent requests are sent to the disk, it is often the case the several requests will require data from the same track. This algorithm will service those requests that require its current track for one full rotation and then move to the next closest track in its current direction. Due to this all requests will cause one full track rotation to be read by the head and arm stickiness will not occur – which could cause unnecessarily long I/O wait times.

LOOK and C-LOOK

The LOOK algorithm is a slightly enhanced version of the SCAN algorithm however rather than having the head move all the way to the outer or inner most track unnecessarily, the disk head only moves in it's current direction until it has serviced the last request in that direction. The head will then switch directions and service all jobs in the opposite direction.

Similarly, C-LOOK (Circular LOOK) is just an enhanced implementation of the C-SCAN algorithm. The head will only move in one direction until it has serviced the last request in that direction, at which point it will move back up to the first job starting from the opposite side.

These to implementations are likely how the SCAN and C-SCAN algorithms are actually implemented in practice as they are significantly more efficient. The head does not need to waste time travelling to the extreme edges of the disk.

Analysis

It is obvious to see that LOOK and C-LOOK are clear improvements over SCAN and C-SCAN, and that C-LOOK is likely a more fair approach to service ordering as the head will be moving in an evenly distributed pattern. These algorithms will likely be the most efficient under normal loads. However when considering the possibility of heavier loads, arm stickiness could pose a potential problem.

Under a heavy load, it is foreseeable that many requests will occur in a short amount of time for data that resides on the same track. This will cause those requests that are for data on a different track to be moved towards the end of the queue until all requests for at least the current track are serviced. At the point in which the head moves to the next track it is possible that there are already an abundance of requests for that new track. As such, under heavy loads, arm stickiness should be at least considered and an algorithm that reduces this should be used instead.

Both N-Step-SCAN and FSCAN will minimize arm stickiness, as all new requests while an I/O queue is being processed are added to a different queue. However, this now allows for potential wasted time as the head may be passing by tracks with data requests that are in a different queue.

Due to this, Eschenbach Scheme is likely the most efficient disk scheduling algorithm for disks with heavy loads. Eschenbach scheme allows for multiple requests on the same track

to be serviced for exacly one rotation, at which point it moves to the next closest head in its current direction. This both prevents arm stickiness, while preventing the head from skipping over tracks with data that has been requested. If the request load was not heavy, then the disk is wasting time reading the entire track when it is less likely for there to be multiple requests for that track – in this case, C-LOOK is the more efficient choice.

Works Cited

- Deitel, Harvey M. "Disk Performance Optimization."*An Introduction to Operating Systems*. Reading, MA: Addison-Wesley Pub., 1984. 359-86. Print.

- "DISK SCHEDULING ALGORITHMS."*DISK SCHEDULING ALGORITHMS*. Illinois Institute of Technology, n.d. Web. 18 Feb. 2015. <http://www.cs.iit.edu/~cs561/cs450/disksched/disksched.html>.

- "Memory Hierarchy."*Memory Hierarchy*. Miller Lab @ University of Toronto, n.d. Web. 18 Feb. 2015. <http://dblab.cs.toronto.edu/courses/443/2014/02.memory-hierarchy.html>.

- Stallings, William. "Disk Scheduling."*Operating Systems: Internals and Design Principles*. 8th ed. New Jersey: Pearson Education, 2009. 489-96. Print.

- Weingarten, Allen. "The Eschenbach Drum Scheme."*Communications of the ACM*July 1966: 509-12.*ACM Digital Library*. Web. 19 Feb. 2015.