# Lecture 32 — Disk Scheduling

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

December 27, 2022

# Disk Scheduling

The disk is a very slow device, as far as the CPU is concerned.


We will examine the hardware behind magnetic disks and see why they are so slow and consider algorithms that could be used to speed up operations.

# Magnetic Disks

Unless you have a Solid State Drive, the mass and permanent storage of data in your system is on a magnetic hard disk.

Due to the physical nature of how disk drives work, there is an associated delay with moving to a new location and reading the data there.

Thus, we would like to devise efficient schedules for reading and writing.

We will exclude SSDs from consideration here.

# Magnetic Disks

SSDs do not contain moving disk heads or spinning platters.

An algorithm to schedule the optimal movement of the disk heads is irrelevant.

Read access times from the SSD are consistent and uniform even if data is being requested from random locations.

The SSD disk scheduling algorithm can be as simple as first-come, first-served.

# Disk Scheduling Motivation

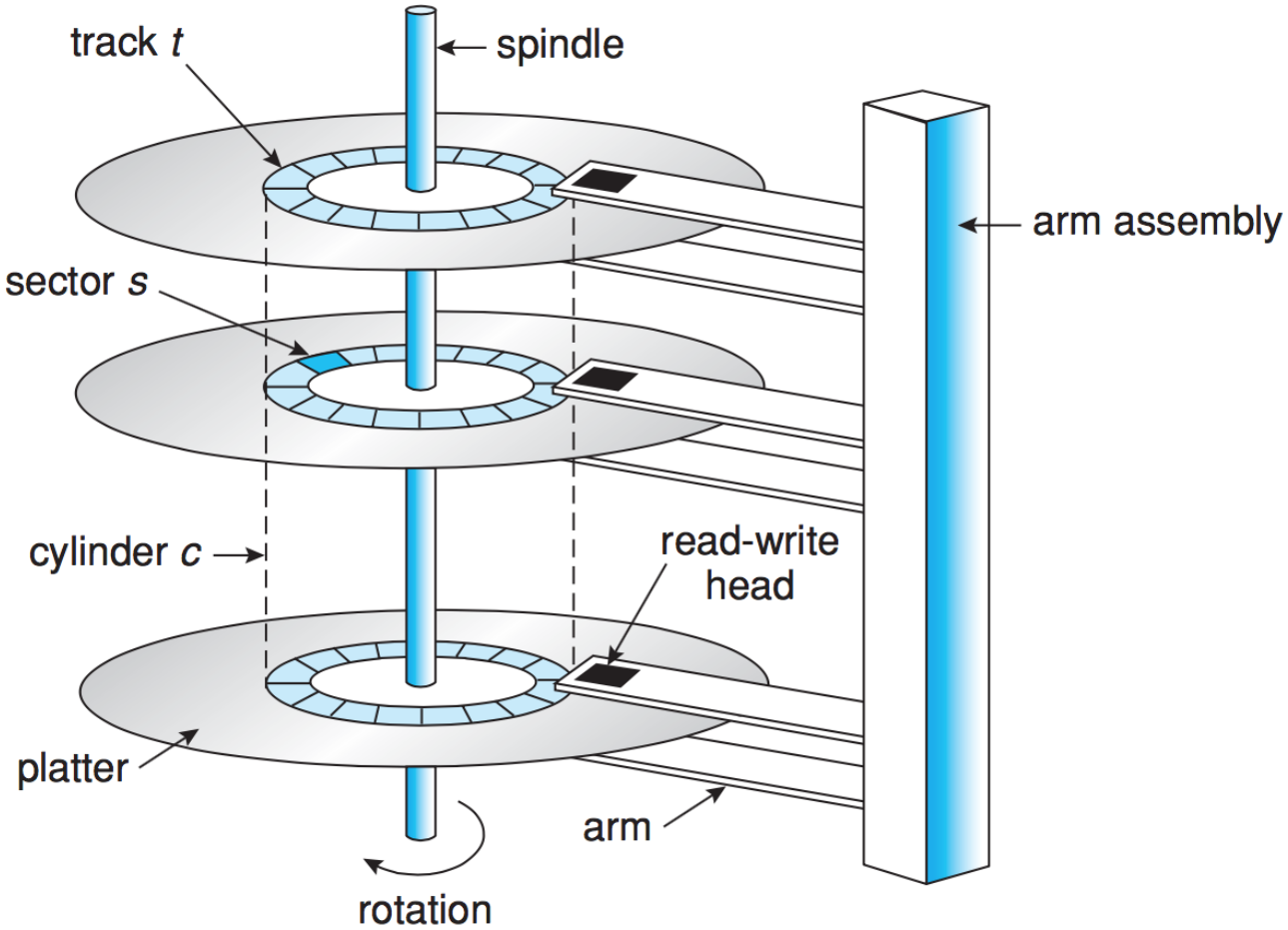
Why bother learning about this if magnetic drives are going away?

The first is that while magnetic drives might be on the way out, they're not dead yet and many systems still use them, including large databases.

The other is that disks are a convenient example, but not the only I/O device that needs to seek around to read and write data.

So what we examine here may be applicable to other sorts of devices too.

# Hard Drive Internals



# Don’t Crash

A read-write head is suspended a very small distance above the surface of each platter and reads or writes data directly beneath it.

This is why they are carefully sealed: if a bit of dust lands on the platter, the read-write head can run into it (which is bad).

It gets even worse if a disk head makes contact with the platter: this is called a head crash and it tends to permanently destroy data.

# Platter Organization

The platter surface is divided logically into different circular tracks, which are respectively divided up into sectors (blocks).

A set of tracks stacked vertically is called a column.

When the disk is in use, a motor spins the platters at high speed and another one manipulates the positions of the arm.

# Disk Performance

The performance of the disk can be broken down into two values.

The first is the transfer rate; the speed at which data can be moved from the disk to the computer or vice-versa.

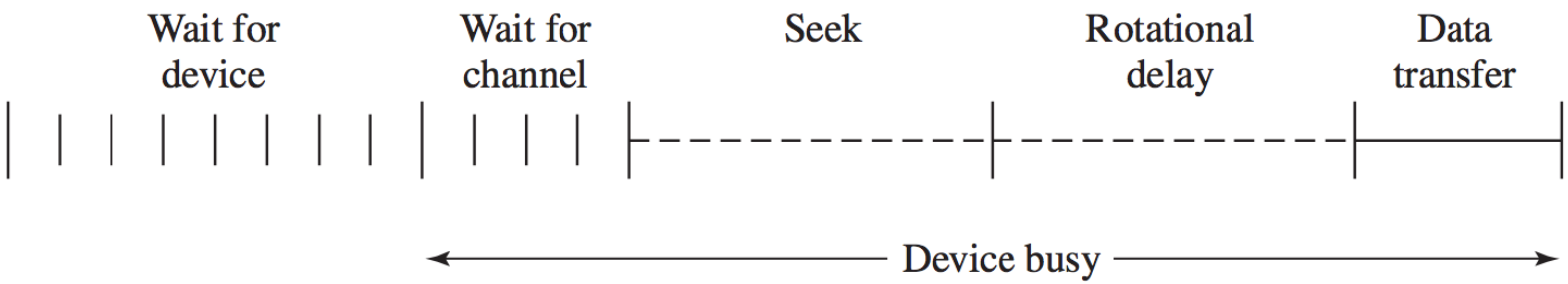
The other is the random-access time; how long it takes to get to a particular piece of data.

The random access time itself is broken down into:

The seek time and the rotational latency

Seek times and rational latencies tend to run in the millisecond range.

# Disk Transfer Time



# Disk Access Time

The total average access time, $T_a$, for a disk operation:

$$T_a = T_s + \frac{1}{2r} + \frac{b}{rN}$$

Where: $T_s$ is the average seek time
$r$ is the rotation speed (revolutions per second)
$b$ is the number of bytes to be transferred
$N$ is the number of bytes on a track.

# Performance Variance

A small example.

The disk has an average seek time of 4 ms, a rotation speed of 7 500 RPM, and 512-byte sectors with 500 sectors per track.

We want to read a file that is 2 500 sectors (a total of 1.28 MB).

# Performance Variance

If the file is stored as compactly on disk as it can be, it is sequentially organized: the file occupies all the sectors on 5 adjacent tracks.

To read the first track, it will take 16 ms: 4 ms to seek, 4 ms rotational delay, and then 8 ms to read 500 sectors.

Because the data is sequential, no additional seek time is necessary, so we just keep reading subsequent sectors which means only the rotational delay.

Each additional track takes 12 ms (4 ms rotational delay + 8 ms to read it). Thus the total time is $16 + (4 \times 12)$ = 64 ms.

# Performance Variance

What if instead the data was randomly distributed on the disk (not sequential)?

The average seek and rotational delay times don't change.

Reading one sector takes 0.016 ms.

So each read of a sector will take a total of 8.016 ms ($4 + 4 + 0.016$).

There are 2 500 sectors so the total is $2\,500 \times 8.016 = 20\,040$ ms.

# THIS READ TIME IS...

# Performance Variance

20 000 ms - twenty full seconds to read 1.28 MB of data? Yes. Seriously.

This speed would be considered utterly unacceptable by the users.

The order in which sectors are read from the disk makes a huge difference.

Decisions about how to store data on disk is extremely important.

Not only is placement important, but how we schedule the reads and writes.

# Disk Scheduling

We should introduce a final metric, the bandwidth.

Bandwidth is the total number of bytes transferred, divided by the total time between the request for service and completion of the transfer.

This is a measure of how much data is effectively transferred in a period of time.

This is one of the measures we would like to improve, as well as the access time.

# Reading or Writing from Disk

When a process needs to read from or write to the disk, the system call contains the following information:

1. If the operation is a read or write.
2. The disk address for the transfer.
3. The memory address for the transfer.
4. How much data to transfer (how many sectors).

# Scheduling Algorithm: Random

Nobody actually seriously advocates random scheduling, but it is a baseline against which to compare various scheduling algorithms.

We could do disk accesses based on process priority, paying no attention to how (in)convenient it is from the perspective of the disk.

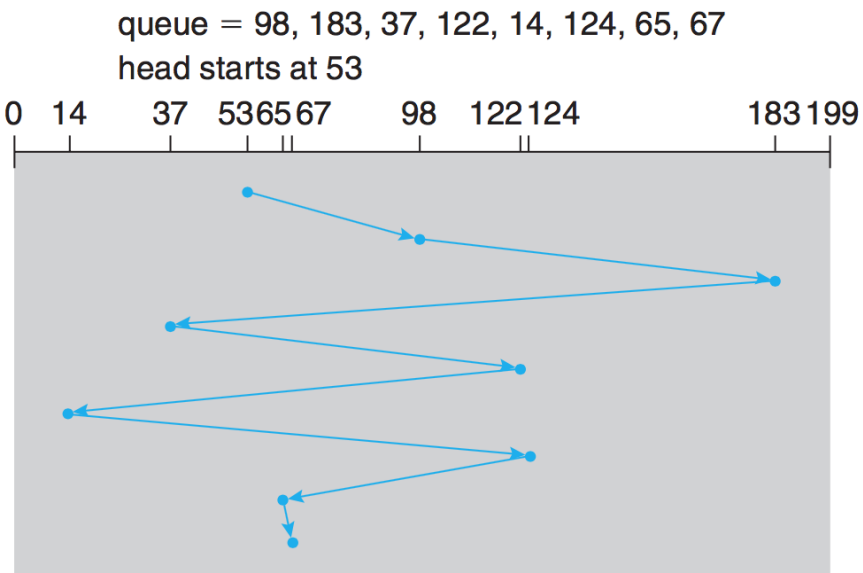
If there is only one request, there is not much of a decision to make.

# First-Come, First-Served

Fair and simple, but does not necessarily provide the fastest service.

No attempt is made to group, organize, or rearrange the requests.

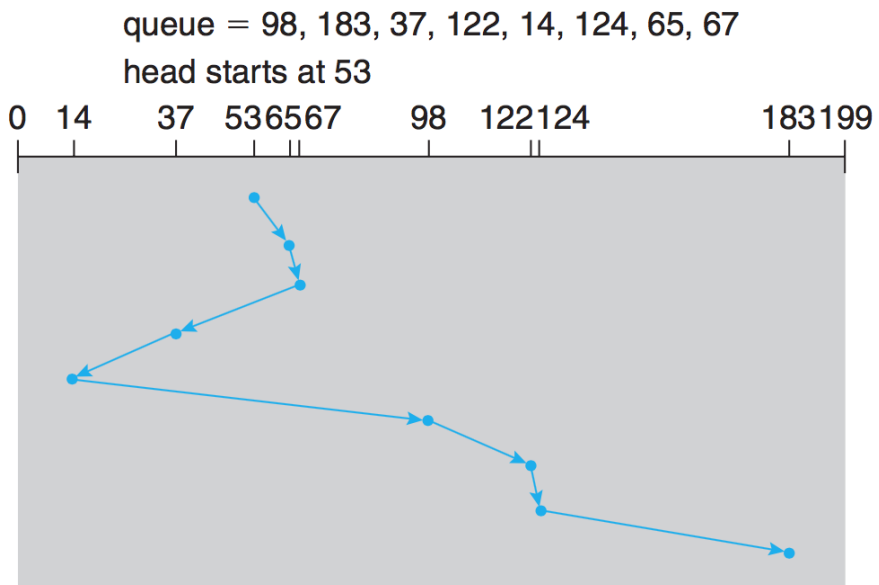
Example: requests for I/O to/from cylinders: {98, 183, 37, 122, 14, 124, 65, 67}; the disk head begins at 53.




Total movement: 640 cylinders.

# Shortest Seek Time First

Choose the request with the least seek time from the current head position.




Total movement: 236 cylinders.

# Shortest Seek Time First

This routine is unfortunately subject to starvation.

Suppose the disk head is at position 14.

While is is there, a new request at 24 arrives, making the request at 98 wait.

If enough low-numbered requests arrive to arrive during execution, a request at a high number may be put off, potentially indefinitely.

The more requests occur, the more likely starvation is.

# Shortest Seek Time First

Though the SSTF algorithm is an improvement, it is not optimal.

In the example, we would reduce the total amount of movement to 208 cylinders if we did the move from 53 to 37, even though it is not the closest.

This algorithm provides for some spatial locality.

# SCAN Scheduling

Move in one direction at a time until it reaches the “end” of the disk.

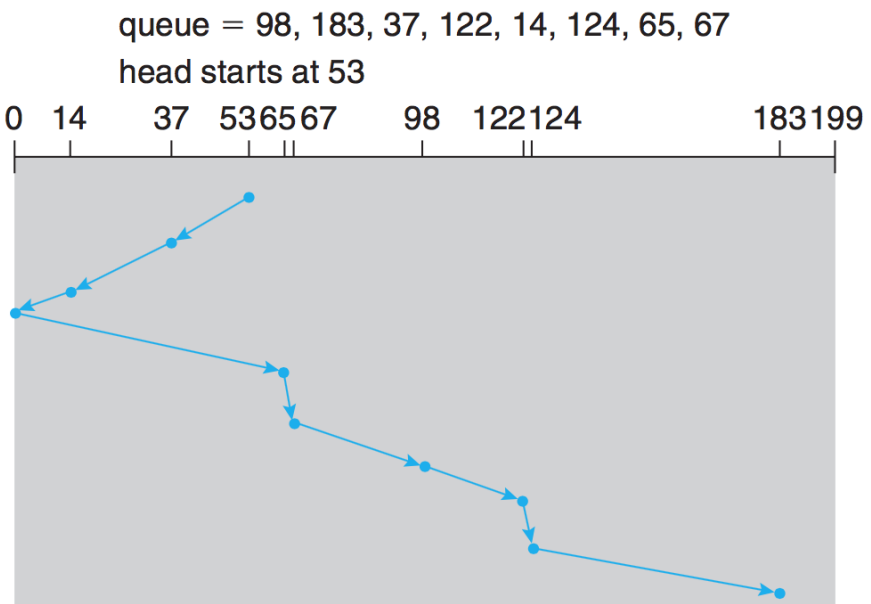
After that, the direction is reversed.

The SCAN algorithm is sometimes called the elevator algorithm.

# SCAN Scheduling

If a request arrives just in front of the head (e.g, 129 arrives before 124 is serviced), it will be serviced virtually right away.

If it is just after where the head has been (e.g., 123 after 124 has been serviced) it will wait until the direction reverses again.



# SCAN Scheduling

The SCAN policy does not take advantage of spatial locality as SSTF does.

If the head has just moved from an area, it can be a long time before it returns.

Thus, if there are multiple accesses in the same area, they are likely to be at least partly spread out in time.

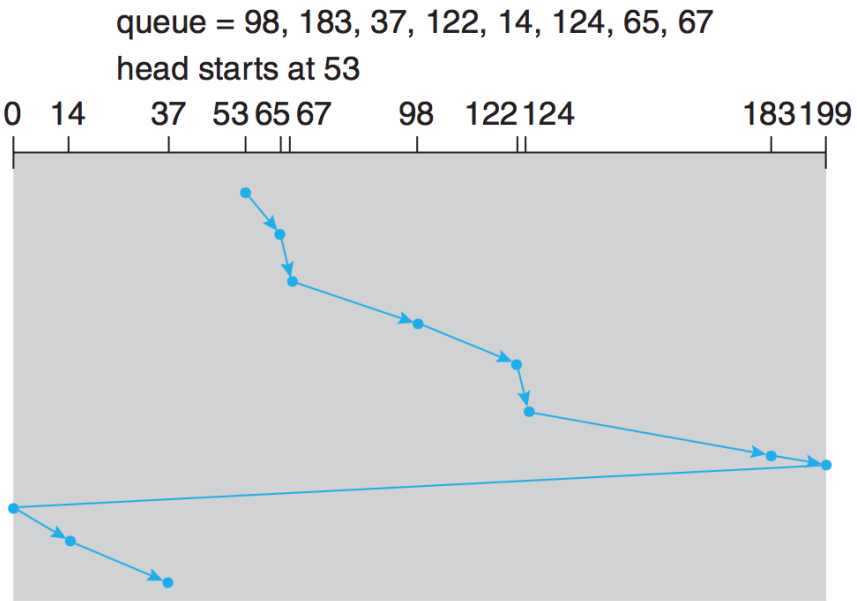
# C-SCAN Scheduling

C-SCAN is designed to exploit the fact that when the disk has just reached one end, most requests are likely at the other end of the disk.

So instead of reversing the direction and servicing requests on the way, jump back to the start of disk immediately and start at the beginning.

It is called C-SCAN because C is for “Circular”, as if the last cylinder just wraps around to the first.

# C-SCAN Scheduling




Assume the amount of time it takes to scan from the start to end of disk is $t$.

The expected service interval for sectors at the periphery is $2t$ with SCAN.

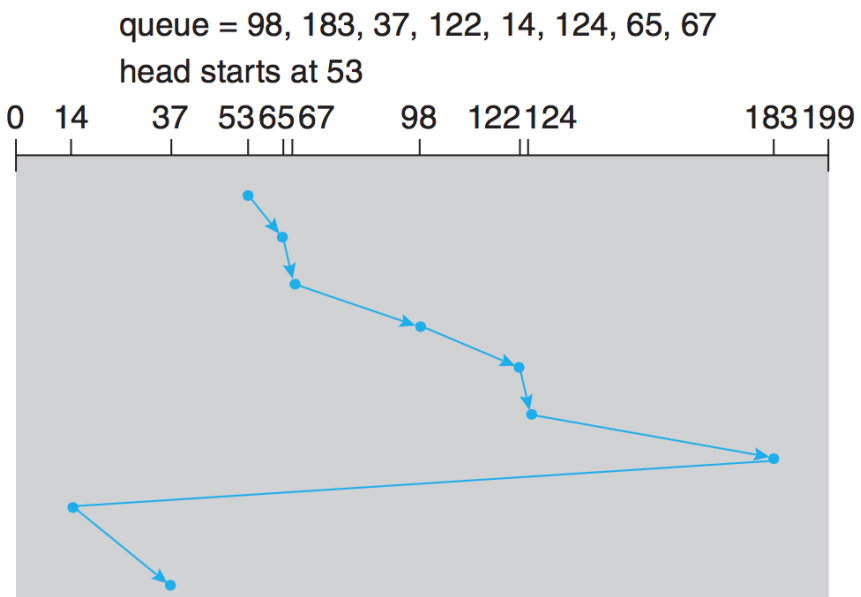
C-SCAN reduces this to $t + s_{max}$ where $s_{max}$ is the maximum seek time.

# LOOK and C-LOOK

The SCAN and C-SCAN algorithms as described can be optimized in a little way.

Instead of going all the way to the end of disk every time, instead just go to the final request and only then reverse direction (or go back to the start).

The names for this variant are LOOK and C-LOOK, respectively.



We can probably agree that FCFS is not the best choice.

How do we decide between the other options?

The LOOK approach seems like it would prevent the starvation problem.

Although we expect that with the SCAN or C-SCAN algorithm, all requests get serviced, it could happen that a process is effectively starved.

# Modification of SCAN

A strategy to prevent this is a modification of the SCAN algorithm such that it has two queues for requests (think double buffering).

While one queue is being emptied, the other is being filled.

Thus, a request will not wait indefinitely; if the queue is of capacity $C$ then any particular read or write will wait, at most, $C$ accesses.

The larger the value of $C$, the more the performance resembles SCAN; a smaller value of $C$ means behaviour is more like FCFS.

The choice of $C$ is a tradeoff; sacrifice performance to increase fairness.

# Scheduling Algorithm Implementation

The scheduling algorithms above consider only the seek times, and not the rotational latency, even though they can be about the same size.

It is very difficult for the OS to schedule for improved rotational latency, because the disk is responsible for the physical placement of the logical blocks.

The hard disk controller takes on some of the scheduling options.

The OS can provide to the controller a grouping of requests, and then the controller will figure out how to schedule them.

# Scheduling Algorithm Choice

If the speed of disk reads and writes were the only thing to be concerned about, the operating system would probably not worry about disk scheduling.

The OS may have certain priorities that should take precedence.

# Scheduling Algorithm Choice

Loading a page into main memory might need to take priority over an application writing a file to disk.

Higher priority processes, especially in real-time systems, should not be waiting (for too long at least) for lower priority processes' disk writes.

So, under some circumstances, the operating system needs to manage the reads and writes and not just leave it up to the disk controller.