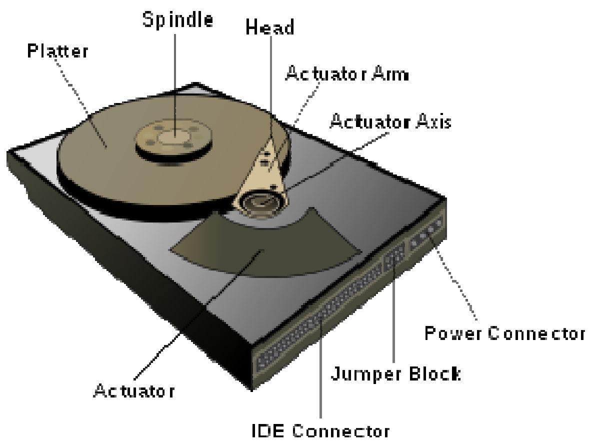
Andrew Hanushevsky: File System I/O

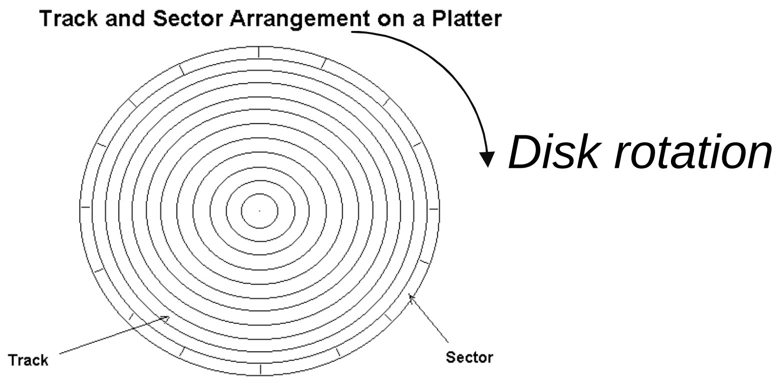
Goals

- Sensitize you to File System limitations
  - How I/O choices make or break performance
- Show what to do and not to do
  - Keeping performance high
- How to broadly translate advice
  - Databases and frameworks

Disk Mechanics

- Disk surface is divided into sectors
  - Usually 512 bytes
- An I/O operation requires that the disk
  - Move the head to the right circular track (seek time)
  - Wait until the proper sector arrives (rotational delay)
  - Then transfer the data





Sources: http://upload.wikimedia.org/wikipedia/commons/thumb/5/52/Hard_drive-en.svg/300px-Hard_drive-en.svg.png
http://www.comptechdoc.org/hardware/pc/begin/hwharddrive.html

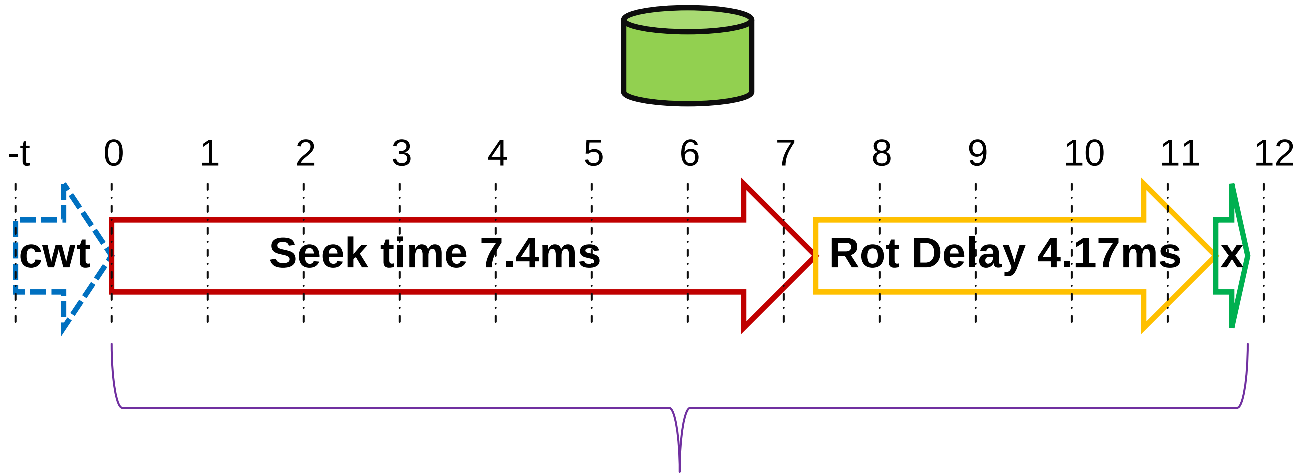
Mechanical Devices Are Slow

Characteristic	Seagate Barracuda 180	Seagate Cheetah X15-36LP	Seagate Barracuda 36ES
Type	High Capacity	High Performance	Desktop
Capacity	181.6GB	36.7GB	18.4GB
Min Seek Time	0.8ms	0.3ms	1.0ms
Avg seek time	7.4ms	3.6ms	9.5ms
Spindle speed	7200rpm	15K rpm	7200 rpm
Avg Rotational Delay	4.17ms	2 ms	4.17 ms
Max xfr rate	160 MB/s	522-709 MB/s	25 MB/s
Sector Size	512	512	512

Source: ftp://ftp.prenhall.com/pub/esm/sample_chapters/engineering_computer_science/stallings/coa6e/pdf/ch6.pdf

Slowness In Perspective

Seagate Barracuda 180



Reading 64K requires, on average, 11.77ms,
excluding channel wait time (cwt).
Actual data transfer (x) occupies disk 1.7% of the total time.
You need to read almost 2MB to achieve 50% channel utilization!
The faster Cheetah drive accomplishes this 48% faster (5.652 ms)
But channel utilization drops to less than 1%
requiring a read of 3MB for 50% channel utilization.

File System Mechanics

- File System groups N sectors into an I/O Unit
  - Usually 8 to 256 sectors (4K to 128K, sometimes more)
- Data always read & written in I/O units or blocks
  - Simplifies mapping files into memory
    - This is why a block size is typically a multiple of the page size
- Data, in unit sizes, is cached in memory
  - Speeds future access to data within the block
- Additional subsequent blocks may be pre-read
  - With the hope they will be wanted in the future

File System & Slowness

- File system tries to hide disk slowness
  - Memory caching to avoid disk I/O
    - Also done in high-end disk controller caches
  - Pre-reading to keep channel utilization high
    - Done in the background to minimize impact
      - Also done in some high-end RAID disk controllers
  - Offset ordering
    - Reduces seek time
      - Also done in high-end disk controllers

File System Performance Varies

Operation	Ext3	Ext4	Improvement
Creation of eight 1 GB files	155.9 sec	145.1 sec	6.9 %
Write speed	55.4 MB/sec	59.3 MB/sec	7.0 %
Deletion of eight 1 GB files	11.87 sec	0.33 sec	97.2 %
10,000 random reads and writes in an 8 GB file	80.0 ops/sec	88.7 ops/sec	10.9 %

Source: http://www.h-online.com/open/The-Ext4-Linux-file-system--/features/113403/1

What This Implies

- FS performance ≈ Disk Performance
- Behavior of application is the determinant
  - How much application data per I/O request?
  - Sequential access?
  - Random access?
    - What is the r/w cycle length?
      - How many different blocks will be hit before a block revisit?
- All of these have a profound effect
  - Independent of file system or disk device
    - These might make it a *little* better or worse

Effect of I/O Request Size

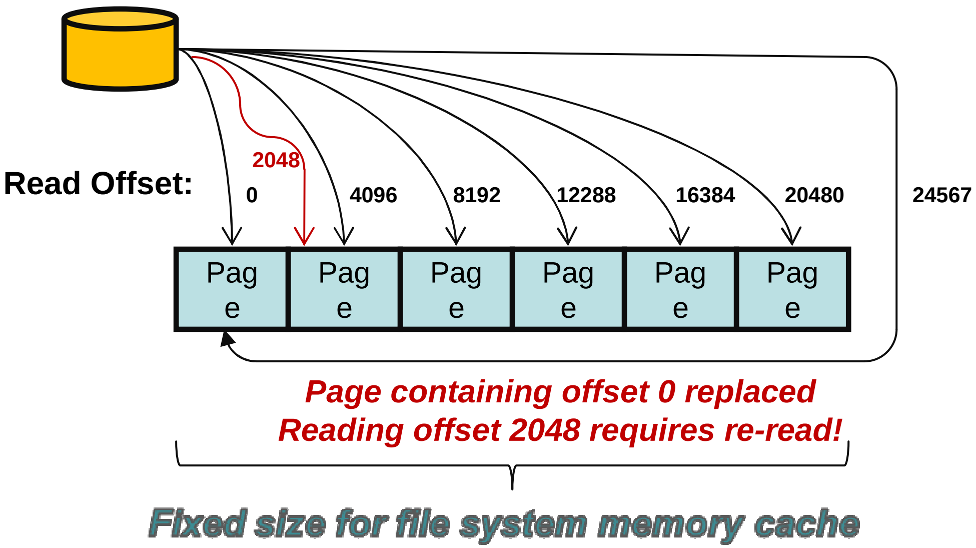
- Recall FS reads/writes data in blocks
  - Assume block is 4K then reading…
    - 512 bytes = 12.5% efficiency
    - 1024 bytes = 25.0% efficiency
    - 2048 bytes = 50.0% efficiency
    - 4096 bytes = 100 % efficiency
  - Application should try to keep efficiency high
    - Each read should be as large as possible
    - Subsequent reads should use cached data

Effect of Sequential Access

- This is the easiest for FS and Disk
  - Large disk devices are inherently sequential
- However, your app is not alone
  - Application interleaving produces random I/O
    - FS read-ahead attempts to alleviate some of this
- Each sequential request should be large
  - 1-4MB per request usually the works best

Effect of Random I/O

- Random I/O is a performance destroyer
  - True for mechanical disks but not for SSD's
- Cycle length is important



Read Offset
0
4096
8192
12288
16384
20480
24567
2048

Cycle Length of 7 pages > cache size of 6 pages

Contrived Example?

- Yes, memory caches are very large today
  - Typically, 1,048,576 pages (4GB)
- Works great if your application is alone
  - Not true for multi-core batch nodes
    - Can shrink to 131,072 pages (500MB) for 8 cores
      - Effective size per running job
  - Definitely not true for file servers
    - They serve thousands of simultaneous clients

Advising The File System

- Can use **posix_fadvise()**
  - Tells file system how the app will access data
    - Starting at an offset for some number of bytes
    - Allows file system to better manage the memory cache
  - Few OS's support this API
    - Not present in MacOS X 10.3, FreeBSD 6.0, NetBSD 3.0, OpenBSD 3.8, AIX 5.1, HP-UX 11, IRIX 6.5, OSF/1 5.1, Solaris 10, Cygwin 1.5.x, mingw, Interix 3.5, BeOS.
    - Present in others but ignored (e.g.,OpenSolaris)
  - So, for now, consider it Linux specific
    - Or using HP/UX 11.31

posix_fadvise() Details

```
#include <fcntl.h>
int posix_fadvise(int fd, off_t offset, off_t len, int advice);
```

Advice:

`POSIX_FADV_NORMAL`

Standard processing

`POSIX_FADV_SEQUENTIAL`

Doubles the read-ahead size for entire file

`POSIX_FADV_RANDOM`

Disables read-ahead for entire file

`POSIX_FADV_WILLNEED`

Initiates block read for specified byte range (also see **readahead()**)

`POSIX_FADV_DONTNEED`

Discards cached file pages in specified byte range

`POSIX_FADV_NOREUSE`

Data will not be used again

Problematic, some OS's support this, some ignore it (e.g., Linux)

If It Were So Simple. . .

- Application data framework complications
  - Databases like mySQL
  - Persistency frameworks like root
- Most HEP applications use one or more
  - Actual disk device is hidden
  - Hard if not impossible to directly apply advice
- What to do?

Databases & Performance

- Translate advice to schema development
  - Avoid wide tables when not needed
    - Increases payload of only some data wanted
  - Use indices for sparsely accessed rows
    - Allows database to optimize access
  - Normalize the tables within reason
    - Keep related data together

Frameworks & Performance

- Know how framework lays out data
  - This is the most difficult part
    - Consult framework experts
  - Carefully construct your data objects
    - Keep useful payload as large as possible
      - Be cognizant of any compression done by framework
  - Cluster related payloads as much as possible
  - Avoid scattered references
    - This reduces widely spaced random reads

Conclusions

- Be aware you're dealing with mechanics
  - Disks are slow and unwieldy devices
- Overlap I/O and CPU as much as possible
  - Choose algorithms that make this possible
    - This also requires deft multi-threading
- Carefully layout your data
  - Keep in mind the database and framework
    - Use the advice in this section