Persistent Storage

Persistent storage

just like memory, only different

- Just like diamonds
  - last forever (?)
    - memory is volatile
  - very dense
    - 1 TByte of storage fits here 
- ...but much cheaper
  - 1 TByte is about $100 on Amazon
    - way cheaper than 

How persistent storage affects OS Design

Goal	Physical Characteristics	Design Implication
High performance		
Named data		
Controlled Sharing		
Reliability		

How persistent storage affects OS Design

Goal	Physical Characteristics	Design Implication
High performance	Large cost to initiate I/O	
Named data		
Controlled Sharing		
Reliability		

How persistent storage affects OS Design

Goal	Physical Characteristics	Design Implication
High performance	Large cost to initiate I/O	Organize storage to access data in large sequential units Use caching
Named data		
Controlled Sharing		
Reliability		

How persistent storage affects OS Design

Goal	Physical Characteristics	Design Implication
High performance	Large cost to initiate I/O	Organize storage to access data in large sequential units Use caching
Named data	Large capacity Survives crashes Shared across programs	
Controlled Sharing		
Reliability		

How persistent storage affects OS Design

Goal	Physical Characteristics	Design Implication
High performance	Large cost to initiate I/O	Organize storage to access data in large sequential units Use caching
Named data	Large capacity Survives crashes Shared across programs	Support files and directories with meaningful names
Controlled Sharing		
Reliability		

How persistent storage affects OS Design

Goal	Physical Characteristics	Design Implication
High performance	Large cost to initiate I/O	Organize storage to access data in large sequential units Use caching
Named data	Large capacity Survives crashes Shared across programs	Support files and directories with meaningful names
Controlled Sharing	Device may store data from many users	
Reliability		

How persistent storage affects OS Design

Goal	Physical Characteristics	Design Implication
High performance	Large cost to initiate I/O	Organize storage to access data in large sequential units Use caching
Named data	Large capacity Survives crashes Shared across programs	Support files and directories with meaningful names
Controlled Sharing	Device may store data from many users	Include with files metadata for access control
Reliability		

How persistent storage affects OS Design

Goal	Physical Characteristics	Design Implication
High performance	Large cost to initiate I/O	Organize storage to access data in large sequential units Use caching
Named data	Large capacity Survives crashes Shared across programs	Support files and directories with meaningful names
Controlled Sharing	Device may store data from many users	Include with files metadata for access control
Reliability	Crash can occur during updates Storage devices can fail Flash memory wears out	

How persistent storage affects OS Design

Goal	Physical Characteristics	Design Implication
High performance	Large cost to initiate I/O	Organize storage to access data in large sequential units Use caching
Named data	Large capacity Survives crashes Shared across programs	Support files and directories with meaningful names
Controlled Sharing	Device may store data from many users	Include with files metadata for access control
Reliability	Crash can occur during updates Storage devices can fail Flash memory wears out	Use transactions Use redundancy to detect and correct failures Migrate data to even the wear

How persistent storage affects applications

- Example: Word processor with auto-save feature
- If file is large and developer is naive
  - poor performance
    - may have to overwrite entire file to write a few bytes!
      - clever doc format may transform updates in appends
  - corrupt file
    - crash while overwriting file
  - lost file
    - crash while copying new file to old file location

The abstraction stack

- I/O systems are accessed through a series of layered abstractions

Application
Library
File System
Block Cache
Block Device Interface
Device Driver
Memory-mapped I/O, DMA, Interrupts
Physical Device

File System API & Performance (Library, File System, Block Cache)

Device Access (Block Device Interface, Device Driver, Memory-mapped I/O, DMA, Interrupts)

The abstraction stack

- I/O systems are accessed through a series of layered abstractions

Application
Library
File System
Block Cache
Block Device Interface
Device Driver
Memory-mapped I/O, DMA, Interrupts
Physical Device

File System API & Performance

Device Access

Block Cache:
- Caches recently read blocks
- Buffers recently written blocks
- Serves as synchronization point
  - ensures a block is only fetched once

Prefetching
- \+ Reduces latency
- \+ Makes one BIG request of many small ones
- \+ Can leverage hardware parallelism
- – Cache pressure
- – I/O contention
- – Wasted effort

The abstraction stack

- I/O systems are accessed through a series of layered abstractions

Application
Library
File System
Block Cache
Block Device Interface
Device Driver
Memory-mapped I/O, DMA, Interrupts
Physical Device

File System API & Performance

Device Access

Block Device Interface:
- Allows data to be read or written in fixed-sized blocks
- Uniform interface to disparate devices

The abstraction stack

- I/O systems are accessed through a series of layered abstractions

Application
Library
File System
Block Cache
Block Device Interface
Device Driver
Memory-mapped I/O, DMA, Interrupts
Physical Device

File System API & Performance

Device Access

Device Driver:
- Translate between OS abstractions and hw-specific details of I/O devices

The abstraction stack

- I/O systems are accessed through a series of layered abstractions

Application
Library
File System
Block Cache
Block Device Interface
Device Driver
Memory-mapped I/O, DMA, Interrupts
Physical Device

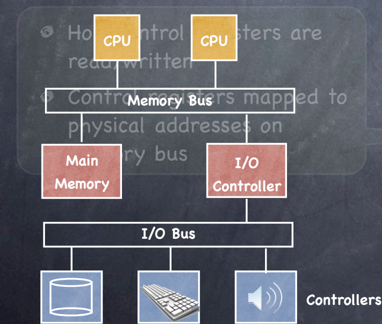
File System API & Performance (Library, File System, Block Cache)

Device Access (Block Device Interface, Device Driver, Memory-mapped I/O, DMA, Interrupts)

- How control registers are read/written
- Control registers mapped to physical addresses on memory bus

The abstraction stack

- I/O systems are accessed through a series of layered abstractions



Application
Library
File System
Block Cache
Block Device Interface
Device Driver
Memory-mapped I/O, DMA, Interrupts
Physical Device

File System API & Performance

Device Access

Audio Controller
Keyboard Controller
Disk Controller

Physical Address Ranges

DRAM

The abstraction stack

- I/O systems are accessed through a series of layered abstractions

Application
Library
File System
Block Cache
Block Device Interface
Device Driver
Memory-mapped I/O, DMA, Interrupts
Physical Device

File System API & Performance

Device Access

- Bulk data transfer between device memory and main memory
- Could be setup using memory mapped I/O
- Target frames pinned until transfer completes

The abstraction stack

- I/O systems are accessed through a series of layered abstractions

Application
Library
File System
Block Cache
Block Device Interface
Device Driver
Memory-mapped I/O, DMA, Interrupts
Physical Device

File System API & Performance

Device Access

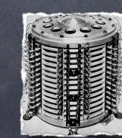
- Notify OS of important events
- Preferable to polling

Example: reading from disk

- Process issues read() system call
- OS moves calling thread to wait queue
- Through memory mapped I/O, OS
  - notifies disk to read requested data
  - sets up DMA to place data in kernel's memory
- Disk reads, performs DMA transfer, triggers interrupt
- Handler:
  - copies data from kernel's memory to user memory
  - moves thread from wait to ready queue
- When thread runs, system call returns with desired data

Storage devices

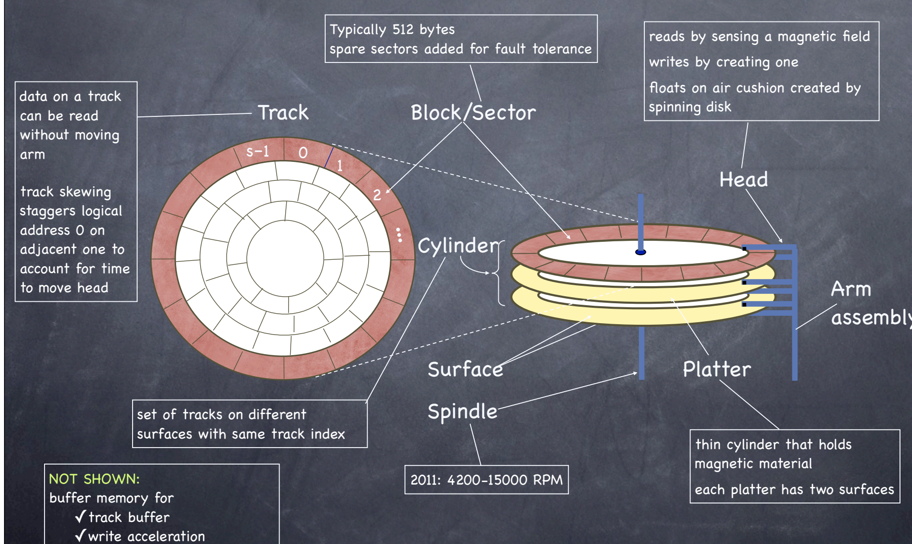
- We focus on two types of persistent storage
  - magnetic disks
    - servers, workstations, laptops
  - flash memory
    - smart phones, tablets, cameras, laptops (right Brian?)
- Other exist(ed)
  - tapes
  - drums
  - clay tablets



Magnetic disk

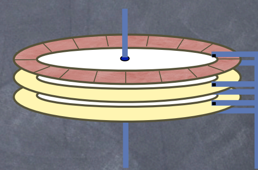
- Store data magnetically on thin metallic film bonded to rotating disk of glass, ceramic, or aluminum

Disk Drive Schematic



Disk Read/Write

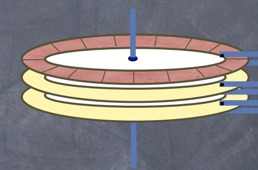
- Present disk with a sector address
  - Old: DA = (drive, surface, track sector)
  - New: Logical Block Address (LBA)
    - linear addressing 0…N-1
- Heads move to appropriate track
  - seek
  - settle
- Appropriate head is enabled
- Wait for sector to appear under head
  - rotational latency
- Read/Write sector
  - transfer time



Disk access time:

Disk Read/Write

- Present disk with a sector address
  - Old: DA = (drive, surface, track sector)
  - New: Logical Block Address (LBA)
    - linear addressing 0…N-1
- Heads move to appropriate track
  - seek
  - settle
- Appropriate head is enabled
- Wait for sector to appear under head
  - rotational latency
- Read/Write sector
  - transfer time

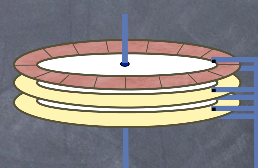


Disk access time:

seek time +

Disk Read/Write

- Present disk with a sector address
  - Old: DA = (drive, surface, track sector)
  - New: Logical Block Address (LBA)
    - linear addressing 0…N-1
- Heads move to appropriate track
  - seek
  - settle
- Appropriate head is enabled
- Wait for sector to appear under head
  - rotational latency
- Read/Write sector
  - transfer time



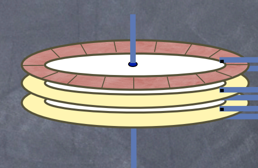
Disk access time:

seek time +

rotation time +

Disk Read/Write

- Present disk with a sector address
  - Old: DA = (drive, surface, track sector)
  - New: Logical Block Address (LBA)
    - linear addressing 0…N-1
- Heads move to appropriate track
  - seek
  - settle
- Appropriate head is enabled
- Wait for sector to appear under head
  - rotational latency
- Read/Write sector
  - transfer time



Disk access time:

seek time +

rotation time +

transfer time

A closer look: seek time

- minimum: time to go from one track to the next
  - 0.3-1.5 ms
- maximum: time to go from innermost to outermost track
  - more than 10ms; up to over 20ms
- average: average across seeks between each possible pair of tracks
  - approximately time to seek 1/3 of the way across disk
  - often pessimistic estimate of performance one observes on actual workload
- head switch time: time to move from track i on one surface to the same track on a different surface
  - range similar to minimum seek time

A closer look: rotation time

- Today most disk rotate at 4200 to 15000 RPM
  - 15ms to 4ms per rotation
  - good estimate for rotational latency is half that amount
- Head starts reading as soon as it settles
  - track buffering to avoid "shoulda coulda"

A closer look: transfer time

- surface transfer time
  - time to transfer one or more sequential sectors to/from surface after head reads/writes first sector
  - much smaller that seek time or rotational latency
    - 512 bytes at 100MB/s ≈ 5μs (0.005 ms)
  - higher for outer tracks than inner ones
    - same RPM, but more space
- host transfer time
  - time to transfer data between host memory and disk buffer
    - 60MB/s (USB) to 2.5GB/s (Fibre Channel 20GFC)

Example: Toshiba MK3254GSY

Size	
Platters/Heads	2/4
Capacity	320GB
Performance	
Spindle speed	7200 RPM
Avg. seek time R/W	10.5/12.0 ms
Max. seek time R/W	19 ms
Track-to-track	1 ms
Surface transfer time	54-128 MB/s
Host transfer time	375 MB/s
Buffer memory	16MB
Power	
Typical	16.35 W
Idle	11.68 W

500 random reads

Size	
Platters/Heads	2/4
Capacity	320GB
Performance	
Spindle speed	7200 RPM
Avg. seek time R/W	10.5/12.0 ms
Max. seek time R/W	19 ms
Track-to-track	1 ms
Surface transfer time	54-128 MB/s
Host transfer time	375 MB/s
Buffer memory	16MB
Power	
Typical	16.35 W
Idle	11.68 W

- Workload
  - 500 read requests, randomly chosen sector
  - served in FIFO order
- How long to service them?
  - seek time: 10.5 ms (avg)
  - rotation time:
    - 7200 RPM = 1/120 RPS
    - rotation time 8.3 ms
    - on average, half of that: 4.15 ms
  - transfer time
    - at least 54 MB/s
    - 512 bytes transferred in (.5/54000) seconds = 9.25μs
  - Total time:
    - 500 x (10.5 + 4.15 + 0.009) ≈ 7.33 sec

500 sequential reads

Size	
Platters/Heads	2/4
Capacity	320GB
Performance	
Spindle speed	7200 RPM
Avg. seek time R/W	10.5/12.0 ms
Max. seek time R/W	19 ms
Track-to-track	1 ms
Surface transfer time	54-128 MB/s
Host transfer time	375 MB/s
Buffer memory	16MB
Power	
Typical	16.35 W
Idle	11.68 W

- Workload
  - 500 read requests for sequential sectors on the same track
  - served in FIFO order
- How long to service them?
  - seek time: 10.5 ms (avg, since don't know where we are starting from)
  - rotation time:
    - 4.15 ms, as in previous example
  - transfer time
    - outer track: 500 x (.5/128000) ≈ 2ms
    - inner track: 500 x (.5/54000) seconds ≈ 4.6ms
  - Total time is between:
    - outer track: (2 + 4.15 + 10.5) ms ≈ 16.65 ms
    - inner track: (4.6 + 4.15 + 10.5) ms ≈ 19.25 ms

500 sequential reads
track buffering edition

Size	
Platters/Heads	2/4
Capacity	320GB
Performance	
Spindle speed	7200 RPM
Avg. seek time R/W	10.5/12.0 ms
Max. seek time R/W	19 ms
Track-to-track	1 ms
Surface transfer time	54-128 MB/s
Host transfer time	375 MB/s
Buffer memory	16MB
Power	
Typical	16.35 W
Idle	11.68 W

- Transfer time
  - outer track ≈ 1/4 of rotation time
  - inner track ≈ 1/2 of rotation time
- Good chance that head, after settling, will be on portion of track it should eventually read
- How good?
  - outer track: 1/4
  - inner track 1/2
- When head is on "good" portion of track, will on average be able to buffer half of it
  - outer track: overlaps transfer time with 1/8 of rotation time. Savings: 1/4 x (1/8 x 8.3 ms) ≈ 0.26 ms
  - inner track: overlaps transfer time with 1/4 of rotation time. Savings: 1/2 x (1/4 x 8.3 ms) ≈ 1.04 ms
- Total time is now between:
  - outer track: (2 + 4.15 + 10.5) ms - savings from overlap ≈ (16.65 - 0.26) ms = 16.39 ms
  - inner track: (4.6 + 4.15 + 10.5) ms - savings from overlap ≈ (19.25 - 1.04) ms = 18.21 ms

500 sequential reads
track buffering edition

Size	
Platters/Heads	2/4
Capacity	320GB
Performance	
Spindle speed	7200 RPM
Avg. seek time R/W	10.5/12.0 ms
Max. seek time R/W	19 ms
Track-to-track	1 ms
Surface transfer time	54-128 MB/s
Host transfer time	375 MB/s
Buffer memory	16MB
Power	
Typical	16.35 W
Idle	11.68 W

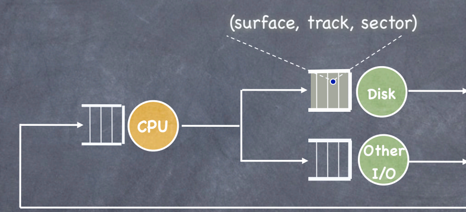
- What fraction of the disk surface bandwidth is achieved?
- Effective bandwidth:
  - (#blocks x size of block)/access time
  - ranges between
    - inner track: 500 x (0.5KB) / (18.21 x 10^{-3} s) = 250/18.21 MB/s ≈ 13.73 MB/s
    - outer track: 500 x (0.5KB) / (16.39 x 10^{-3} s) ≈ 15.25 MB/s
  - as a percentage:
    - inner track: (13.73 MB/s)/54 MB/s = 25.4%
    - outer track: (15.25 MB/s)/128 MB/s = 11.9%

Example: Efficient access

Size	
Platters/Heads	2/4
Capacity	320GB
Performance	
Spindle speed	7200 RPM
Avg. seek time R/W	10.5/12.0 ms
Max. seek time R/W	19 ms
Track-to-track	1 ms
Surface transfer time	54-128 MB/s
Host transfer time	375 MB/s
Buffer memory	16MB
Power	
Typical	16.35 W
Idle	11.68 W

- How large must a read request that begins on a random sector be to achieve at least 80% of max surface transfer bandwidth?
  - To read a sequence of sequential blocks:
    - read entire track
    - do a 1 track seek
    - read next track
  - Track buffering: if we read a full track, starting point does not matter
  - To get 80% of peak bandwidth after a random seek
    - must read enough rotations r to ensure that 80% of time is spent reading
    - 0.8 x total time = r x rotation time
    - 0.8 x (10.5ms + r x (1 + 8.4)ms) = r x 8.4ms
    - r = (8.4ms/0.88ms) = 9.54 rotations
  - Transfer during each rotation: 128 MB/s x 8.4ms = 1.07 MB/s
  - Total transfer: 1.07 MB x 9.54 = 10.2 MB

Disk Head Scheduling



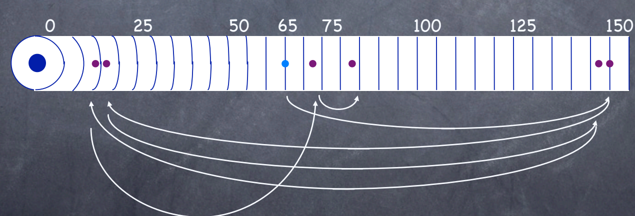
- OS maximizes disk I/O throughput by minimizing head movement through disk head scheduling

FCFS

- Assume a queue of request exists to read/write tracks

83	72	14	147	16	150

and the head is on track 65



FCFS scheduling results in the head moving 550 tracks

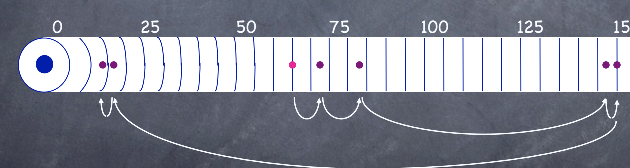
SSTF: shortest seek time first

- Greedy scheduling
  - Rearrange queue from:

83	72	14	147	16	150

  - to:

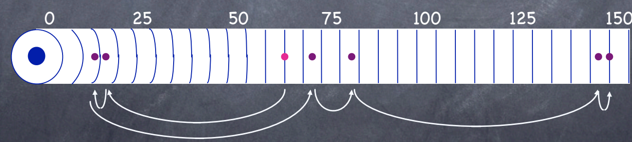
14	16	150	147	83	72



SSTF scheduling results in the head moving 221 tracks

SCAN scheduling

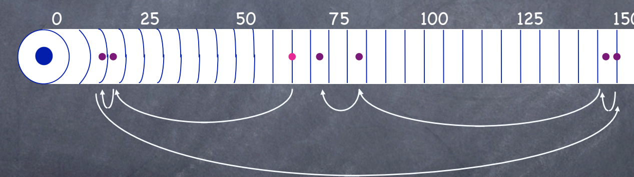
- Move the head in one direction until all requests have been serviced, and then reverse
  - Rearrange queue from: | 83 | 72 | 14 | 147 | 16 | 150 |
  - to: | 150 | 147 | 83 | 72 | 14 | 16 |



Head moves 187 tracks.

C-SCAN scheduling

- Circular SCAN
  - move the head in one direction until an edge of the disk is reached and then reset to the opposite edge



Example: Effects on disk scheduling/CSCAN

Size	
Platters/Heads	2/4
Capacity	320GB
Performance	
Spindle speed	7200 RPM
Avg. seek time R/W	10.5/12.0 ms
Max. seek time R/W	19 ms
Track-to-track	1 ms
Surface transfer time	54-128 MB/s
Host transfer time	375 MB/s
Buffer memory	16MB
Power	
Typical	16.35 W
Idle	11.68 W

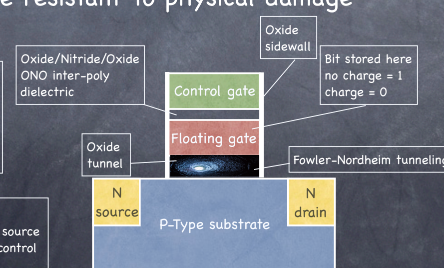
- 500 read requests, to a randomly chosen sector; disk head on outside track; CSCAN
- Seek time
  - average seek for one request ≈ 0.2% across disk
  - estimate as 1-track seek + interpolation with avg seek time
    - 1 + (.2/33.3) X 10.5 ≈ 1.06 ms
- Rotation time
  - We don't know head position when seek ends; random reads
    - 4.15 ms (half rotation)
- Transfer time
  - just as before, at least 9.5 μs
- Total time:
  - 5.22 ms per block
  - For 500 blocks:
    - 500 x 5.22 ms = 2.61 s

Flash storage

- No moving parts
  - better random access performance
  - less power
  - more resistant to physical damage

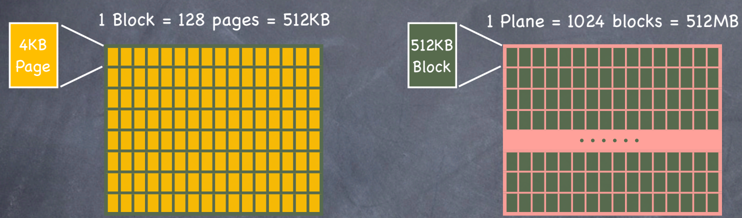
To write 0
- apply positive voltage to drain
- apply even stronger positive voltage to control gate
- electrons are tunneled into floating gate

To write 1
- apply negative voltage to source
- apply positive voltage to control gate
- electrons are forced out of floating gate into source



To read
- apply positive (lower than write)voltage to control gate
- apply positive (lower than write)voltage to drain
- measure current between source and drain to determine whether electrons in gate
  - measured current can encode more than a single bit

NAND flash units



- Operations
  - Erase erasure block
    - before it can be written, needs to be set to logical "1"
    - operation takes several ms
    - Flash translation layer maps logical page to several physical pages; logical page is written to already erased physical page and mapping is adjusted
  - Write page
    - tens of μs
  - Read page
    - tens of μs
- Flash devices can have multiple independent data paths
  - OS can issue multiple concurrent requests to maximize bandwidth

Example: Remapping flash drives

- Flash drive specs
  - 4 KB page
  - 3ms flash
  - 512kB erasure block
  - 50μs read/page/write page
- How long to naively read/erase/and write each page?
  - $128 \times (50 \times 10^{-3} + 50 \times 10^{-3}) + 3 = 15.8$ms
- Suppose we use remapping, and we always have a free erasure block available. How long now?
  - $3/128 + 50 \times 10^{-3} = 73.4\mu s$

read block;
erase;
write entire block

Flash durability

- Flash memory stops reliably storing a bit
  - after many erasures (in the order of 10^3 to 10^6)
  - after a few years without power
  - after nearby cell is read many times (read disturb)
- To improve durability
  - error correcting codes
    - extra bytes in every page
  - management of defective pages/erasure blocks
    - firmware marks them as bad
  - wear leveling
    - spreads updates to hot logical pages to many physical pages
  - spares (pages and erasure blocks)
    - for both wear leveling and managing bad pages and blocks

Example: Intel 710 series Solid State Drive

Size	
Capacity	300GB
Page size	4KB
Performance	
Bandwidth (seq reads)	270 MB/s
Bandwidth (seq writes)	210 MB/s
Read/Write Latency	75μs
Random Reads/sec	38,500 (one every 26 μs)
Random Writes/sec	2,000 (2400 with 20% space reserve)
Interface	SATA 3Gb/s
Endurance	
Endurance	1.1 PB (1.5 PB with 20% space reserve)
Power	
Active	3.7 W
Idle	0.7 W

- Consider 500 read requests to randomly chosen pages. How long will servicing them take?
  - $500 \times 26\mu s = 13$ms
    - spinning disk: 7.8s
- How do random and sequential read performance compare?
  - effective bw random
    - (500×4)KB / 13ms ≈ 154MB/s
  - ratio: 154/270 = 57%
- 500 random writes
  - 500s/2000 = 250ms
- How do random and sequential write compare?
  - effective bw random
    - (500×4)KB / 13ms = 8MB/s
  - ratio: 8/210 = 3.8%

Spinning disk vs flash

Metric	Spinning disk	Flash
Capacity/Cost	Excellent	Good
Sequential BW/Cost	Good	Good
Random I/O per sec/ Cost	Poor	Good
Power Consumption	Fair	Good
Physical Size	Good	Excellent

The File System abstraction

- File system
  - presents applications with persistent, named data
  - a file is a named collection of data. Has two parts
    - data – what a user or application puts in it
      - array of untyped bytes (in MacOS EFS, multiple streams per file)
    - metadata – information added and managed by the os
      - size, owner, security info, modification time
  - a directory provides names for files
    - a list of human readable names
    - a mapping from each name to a specific underlying file or directory (hard link). [A soft link is a mapping from a file name to another file name]
  - path: string that identifies a file or directory
    - absolute (if it stats with "/", the root directory)
    - relative (w.r.t. the current working directory)
  - mount: allows multiple file systems to form a single logical hierarchy
    - a mapping from some path in existing file system to the root directory of the mounted file system

File system API

- Creating and deleting files
  - create() creates a new file with some metadata and a name for the file in a directory
  - link() creates a hard link–a new name for the same underlying file
  - unlink() removes a name for a file from its directory. If last link, file itself and resources it held are deleted
- Open and close
  - open() provides caller with a file descriptor to refer to file
    - permissions checked at open() time
    - creates per file data structure, referred to by file descriptor
      - file ID, R/W permission, pointer to process position in file
  - close() releases data structure
- File access
  - read(), write(), seek()
    - but can use mmap() to create a mapping between region of file and region of memory
  - fsync() does not return until data is written to persistent storage

Block vs Sector

- OS may choose block size larger than a sector on disk.
  - each block consists of consecutive sectors (why?)
    - larger block size increases transfer efficiency (why?)
    - can be handy to have block size equal page size (why?)
  - most systems allow for multi-sector transfer before issuing an interrupt

File system: Functionality and Implementation

- Functionality:
  - File system translates from file name and offset to data block
    - find the blocks that constitute the file
      - must balance locality with expandability
      - must manage free space
    - provide file naming organization
      - e.g. a hierarchical name space
- Implementation:
  - file header (descriptor, inode): owner id, size, last modified time, and location of all data blocks
    - OS should find block number N without accessing disk
      - math, or cached data structure
  - data blocks
    - directory data blocks
      - human readable names, permissions
    - file data blocks
      - data
  - superblocks, group descriptors
    - how large is the file system, how many iNodes, where to find free space, etc.

File system properties

- Most files are small
  - need strong support for small files
  - block size can;t be too big
- Some files are very large
  - must allow large files (64bit file offsets)
  - large file access should be reasonably efficient

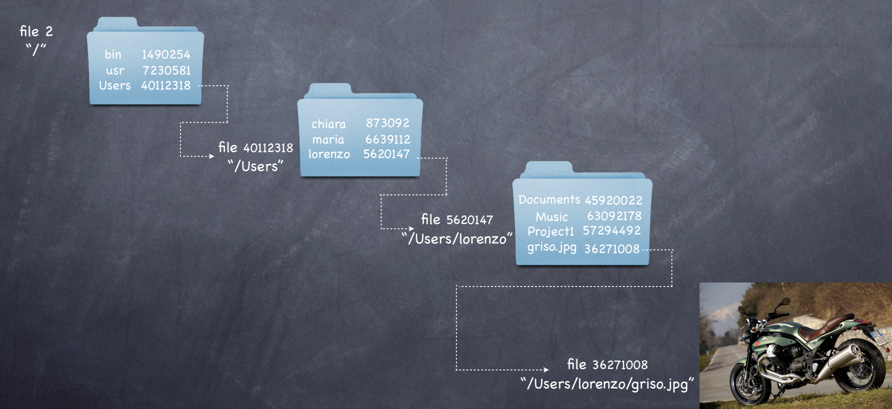
Directory

- A file that contains a collection of mapping from file name to file number

- To look up a file, find the directory that contains the mapping to the file number
- To find that directory, find the parent directory that contains the mapping to that directory's file number...
- Good news: root directory has well-known number (2)

Looking up a file

- Find file /Users/lorenzo/griso.jpg



Finding data

- Index structure provides a way to locate each of the file's blocks
  - usually implemented as a tree for scalability
- Free space map provides a way to allocate free blocks
  - often implemented as a bitmap
- Locality heuristics group data to maximize access performance

FAT File system

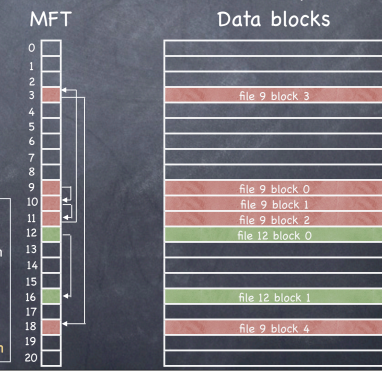
Microsoft, late 70s

- File Allocation Table (FAT)
  - started with MSDOS
  - in FAT-32, supports 2^{28} blocks and files of 2^{32}–1 bytes

Index Structures
Master File Table (MFT)
- array of 32-bit entries
- file represented as a linked list of FAT entries
- file # = index of first FAT entry

Free space map
- If data block i is free, then MFT[i] = 0
- find free blocks by scanning MFT

Locality heuristics
- As simple as next fit:
  - scan sequentially from last allocated entry and return next free entry
- Can be improved through defragmentation



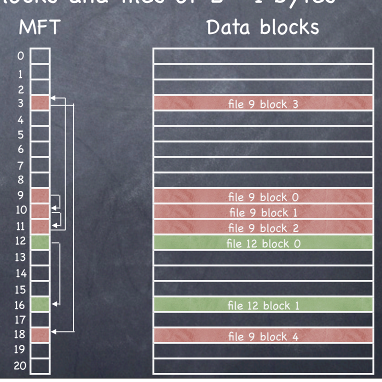
FAT File system

Microsoft, late 70s

- File Allocation Table (FAT)
  - started with MSDOS
  - in FAT-32, supports 2^{28} blocks and files of 2^{32}–1 bytes

Advantages
- simple!
  - used in many USB flash keys
  - used even within MS Word!

Disadvantages
- Poor locality
  - next fit? seriously?
- Poor random access
  - needs sequential traversal
- Limited access control
  - no file owner or group ID metadata
  - any user can read/write any file
- No support for hard links
  - metadata stored in directory entry
- Volume and file size are limited
  - FAT entry is 32 bits, but top 4 are reserved
  - no more than 2^{28} blocks
  - with 4kB blocks, at most 1TB volume
  - file no bigger than 4GB
- No support for transactional updates



FFS: Fast File System

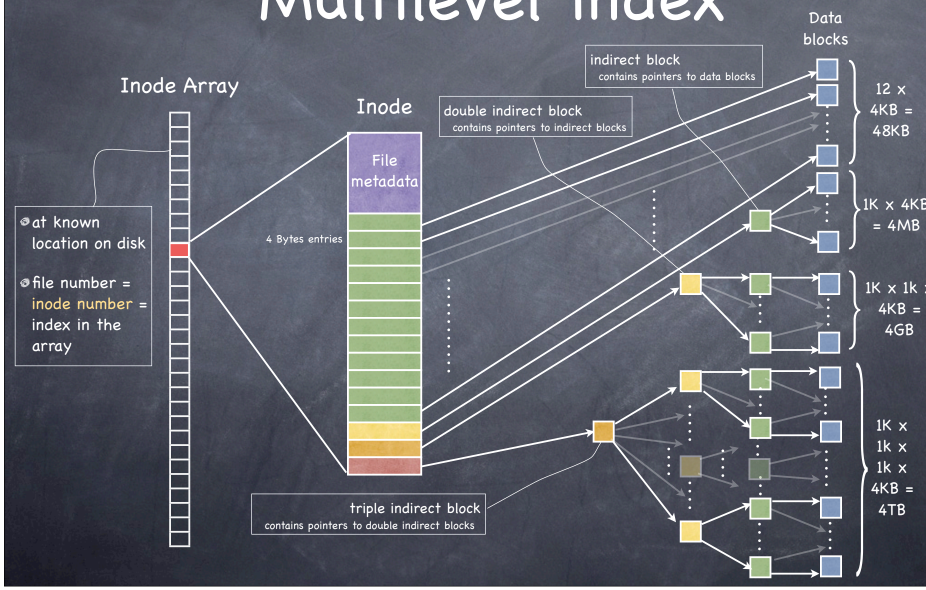
Unix, 80s

- Smart index structure
  - multilevel index allows to locate all blocks of a file
    - efficient for both large and small files
- Smart locality heuristics
  - block group placement
    - optimizes placement for when a file data and metadata, and other files within same directory, are accessed together
  - reserved space
    - gives up about 10% of storage to allow flexibility needed to achieve locality

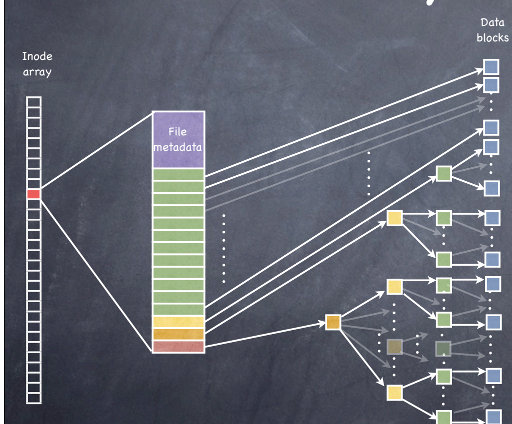
File structure

- Each file is a fixed, asymmetric tree, with fixed size data blocks (e.g. 4KB) as its leaves
- The root of the tree is the file's inode
  - contains file's metadata
    - owner, permissions (rwx for owner, group other), directory?, etc
    - setuid: file is always executed with owner's permission
      - add flexibility but can be dangerous
    - setgid: like setuid for groups
  - contains a set of pointers
    - typically 15
    - first 12 point to data block
    - last three point to intermediate blocks, themselves containing pointers
      - 13: indirect pointer
      - 14: double indirect pointer
      - 15: triple indirect pointer

Multilevel index

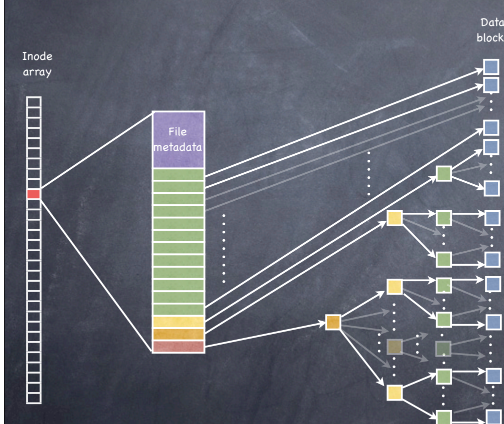


Multilevel index: key ideas



- Tree structure
  - efficient in finding blocks
- High degree
  - efficient in sequential reads
    - once an indirect block is read, can read 100s of data block
- Fixed structure
  - simple to implement
- Asymmetric
  - supports efficiently files big and small

Example: variations on the FFS theme



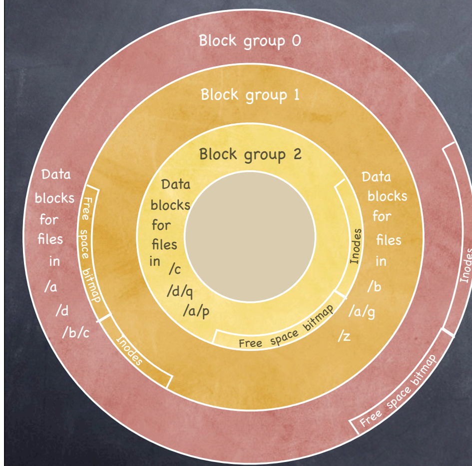
Total = $(256 + .5 + 10^{-6} + 2 \times 10^{-9} + 4.8 \times 10^{-11}) \approx 256.5$ TB

- In BigFS an inode stores
- 4kb blocks, 8 byte pointers
  - 12 direct pointers
  - 1 indirect pointer
  - 1 double indirect
  - 1 triple indirect
  - 1 quadruple indirect
- What is the maximum size of a file?
  - Through direct pointers
    - 12 x 4kb = 48KB
  - Indirect pointer
    - 512 x 4kb = 2MB
  - Double indirect pointer
    - 512^2 x 4kb = 1GB
  - Triple indirect pointer
    - 512^3 x 4kb = 512GB
  - Quadruple indirect pointer
    - 512^4 x 4kb = 256TB

Free space management

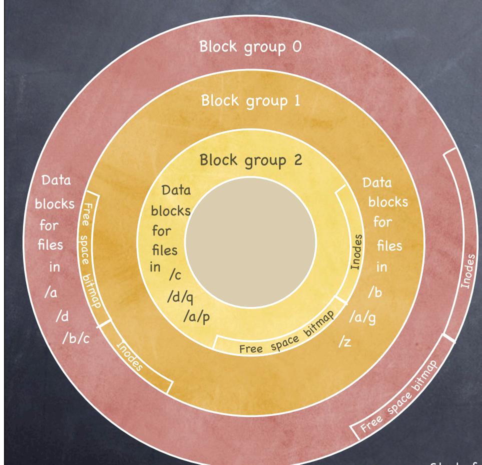
- Easy
  - a bitmap with one bit per storage block
  - bitmap location fixed at formatting time
  - i-th bit indicates whether i-th block is used or free

Locality heuristics: block group placement

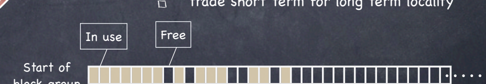


- Divide disk in block groups
  - sets of nearby tracks
- Distribute metadata
  - old design: free space bitmap and inode map in a single contiguous region
    - lots of seeks when going from reading metadata to reading data
  - FFS: distribute free space bitmap and inode array among block groups
- Place file in block group
  - when a new file is created, FFS looks for inodes in the same block as the file's directory
  - when a new directory is created, FFS palces it in a different block from the parent's directory
- Place data blocks
  - first free heuristics
  - trade short term for long term locality

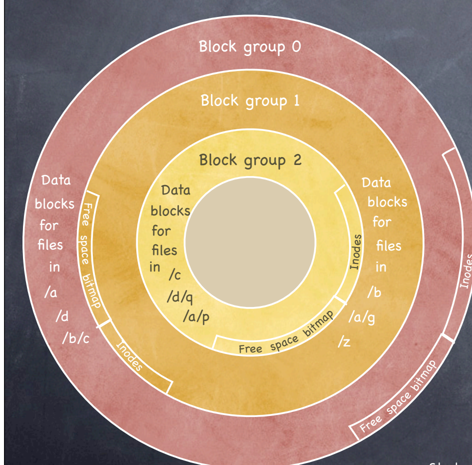
Locality heuristics: block group placement



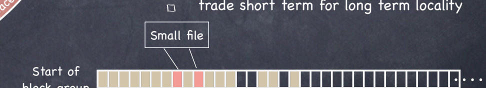
- Divide disk in block groups
  - sets of nearby tracks
- Distribute metadata
  - old design: free space bitmap and inode map in a single contiguous region
    - lots of seeks when going from reading metadata to reading data
  - FFS: distribute free space bitmap and inode array among block groups
- Place file in block group
  - when a new file is created, FFS looks for inodes in the same block as the file's directory
  - when a new directory is created, FFS palces it in a different block from the parent's directory
- Place data blocks
  - first free heuristics
  - trade short term for long term locality



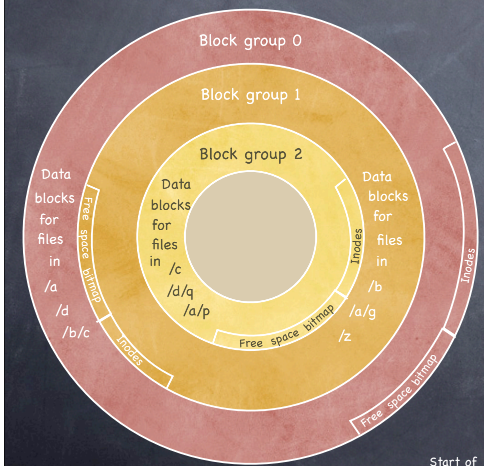
Locality heuristics: block group placement



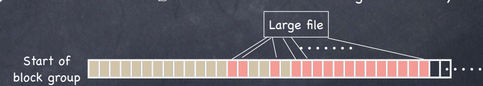
- Divide disk in block groups
  - sets of nearby tracks
- Distribute metadata
  - old design: free space bitmap and inode map in a single contiguous region
    - lots of seeks when going from reading metadata to reading data
  - FFS: distribute free space bitmap and inode array among block groups
- Place file in block group
  - when a new file is created, FFS looks for inodes in the same block as the file's directory
  - when a new directory is created, FFS palces it in a different block from the parent's directory
- Place data blocks
  - first free heuristics
  - trade short term for long term locality



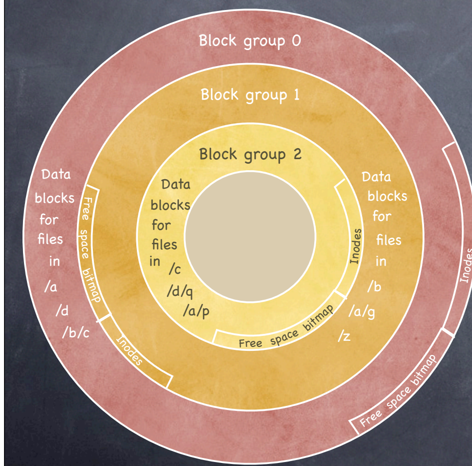
Locality heuristics: block group placement



- Divide disk in block groups
  - sets of nearby tracks
- Distribute metadata
  - old design: free space bitmap and inode map in a single contiguous region
    - lots of seeks when going from reading metadata to reading data
  - FFS: distribute free space bitmap and inode array among block groups
- Place file in block group
  - when a new file is created, FFS looks for inodes in the same block as the file's directory
  - when a new directory is created, FFS palces it in a different block from the parent's directory
- Place data blocks
  - first free heuristics
  - trade short term for long term locality



Locality heuristics: reserved space



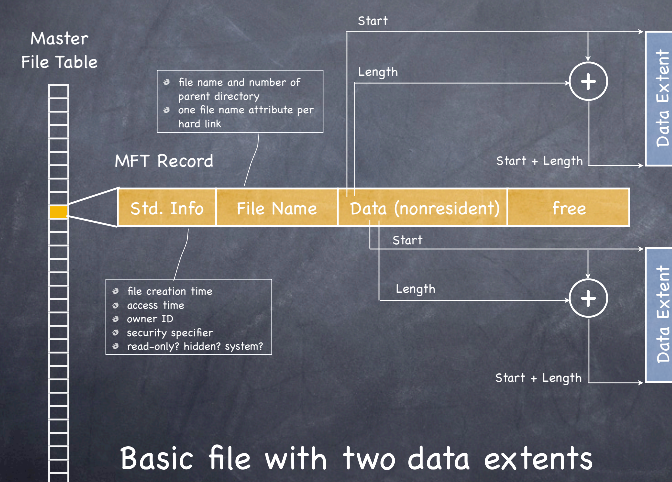
- When a disk is full, hard to optimize locality
  - file may end up scattered through disk
- FFS presents applications with a smaller disk
  - about 10% smaller
  - user write that encroaches on reserved space fails
  - super user still able to allocate inodes

NTFS: flexible tree with extents

Microsoft, 93s

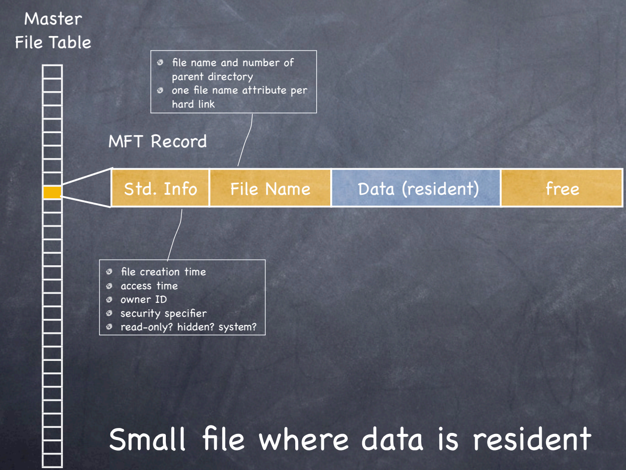
- Index structure: extents and flexible tree
  - extents
    - track ranges of contiguous blocks rather than single blocks
  - flexible tree
    - file represented by variable depth tree
      - large file with few extents can be stored in a shallow tree
  - MFT (Master File Table)
    - array of 1 KB records holding the trees' roots
    - similar to inode table
    - each record stores sequence of variable-sized attribute records
      - both data and metadata are treated as attributes
      - attributes can be resident or nonresident

Example of NTFS index structure



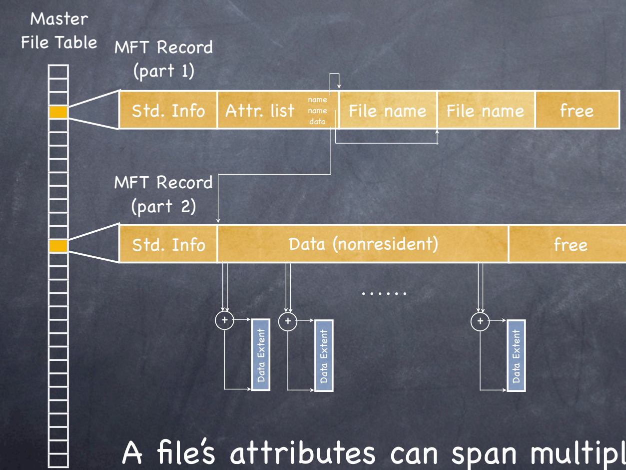
Basic file with two data extents

Example of NTFS index structure



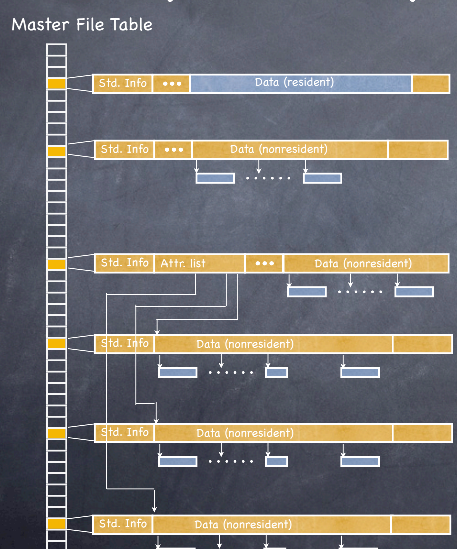
Small file where data is resident

Example of NTFS index structure



A file's attributes can span multiple records

Small, normal, and big files



...and for really huge (or badly fragmented) files, even the attribute list can become nonresident

Metadata files

- NTFS stores most metadata in ordinary files with well-known numbers
  - 5 (root directory); 6 (free space bitmap); 8 (list of bad blocks)
- $Secure (file no. 9)
  - stores access control list for every file
  - indexed by fixed-length key
  - file store appropriate key in their MFT record
- $MFT (file no. 0)
  - stores Master File Table
  - to read MFT, need to know fist entry of MFT
    - a pointer to it stored in first sector of NTFS
  - MFT can start small and grow dynamically
  - To avoid fragmentation, NTFS reserves part of start of volume to MFT expansion
    - when full, halves reserved MFT area

Locality heuristics

- Best fit
  - finds smallest region large enough to fit file
  - NTFS caches allocation status for a small area of disk
    - writes that occur together in time get clustered together
  - SetEnfOfFile() lets specify expected length of file at creation

File access in FFS

- What it takes to read /Users/lorenzo/wisdom.txt
  - Read Inode for "/" (root) from a fixed location
  - Read first data block for root
  - Read Inode for /Users
  - Read first data block of /Users
  - Read Inode for /Users/lorenzo
  - Read first data block for /Users/lorenzo
  - Read Inode for /Users/lorenzo/wisdom.txt
  - Read data blocks for /Users/lorenzo/wisdom.txt

"A cache is a man's best friend"