

Lecture S3: File system – data layout, naming

Review -- 1 min

Intro to I/O

Performance model: Log

Disk physical characteristics/desired abstractions

Physical reality

Desired abstraction

disks are slow

fast access to data

sector addresses (“platter 2, cylinder 42, sector 15”)

named files, directories

write 1 sector at a time

atomic writes, transactions

disks sometimes fail

data lives forever

1.1 Case study: sector layout

What is the fastest way to lay out a sequential file on disk

answer 1:

a series of sequential sectors on a track

problem (in old systems)

read sector 1

process sector 1

read sector 2 -- whoops, sector 2 is already past

wait 1 rotation

read sector 2

...

→ N rotations to read N blocks

→ BW for sequential read is 512 bytes/rotation = 100KB/s

answer 2: (in old systems)

skip 1 sector (or 2 sectors) between sequential blocks

→ 2 rotations to read N blocks

answer 3: (modern systems)
 track buffer -- on-disk cache
 read entire sector into track buffer
 in parallel (once sector 1 arrives...) read sector 1 (from track)
 then read sector 2
 ...
 → 1 rotation to read N blocks

Moral: OS designer needs to understand physical properties of disk

Latency, overhead, bandwidth:

From disk -- what is overhead for a 1-sector read?

what is latency for a 1-sector read?

what is bandwidth term for a 1-sector read?

From CPU/memory system

- what is overhead for a 1-sector read
- what is latency for a 1 sector read
- what is BW term for a 1-sector read

Be careful: What is end-to-end average bandwidth for a 1-sector read
 (people phrase this question to mean end-to-end bytes/sec including
 latency and overhead)

2. Technology trends

1. Disks getting smaller for similar capacity
 smaller → disk spins faster (less rotational delay, higher BW)
 smaller → less distance for head to travel (faster seeks)
 smaller → lighter weight (for portables)
2. disk data getting denser (more bits/square inch; allows smaller
 disks w/o sacrificing capacity)
 Tracks closer together → faster seeks
3. Disks getting cheaper (2x/year since 1991)
4. Disks getting (a little) faster
 seek, rotation – 5-10%/year (2-3x per decade)
 bandwidth – 20-30%/year (~10x per decade)

Overall – disk density (\$/byte) improving much faster than
 mechanical limitations (seek, rotation)

Key to improving density: get head close to surface

Heads are spring loaded, aerodynamically designed to fly as close to surface as possible (also, lightweight to allow for faster seeks)

What happens if head contacts surface? Head crash – scrapes off magnetic material (and data)

Outline - 1 min

Data layout

- (1) given a file header, find the file's blocks
- (2) given a file's name, find its header

mechanism v. policy

Preview - 1 min

File systems

- Performance -- data layout
- Performance/persistence -- naming
- Reliability -- transactions

Networks

Security

Lecture - 20 min

Key abstraction: File
name->data

2 key ideas

- (1) given a file header, find the file's blocks
file ID, offset -> block address

Answer: array, hash, linked list, *tree*

(2) given a file's name, find its header

Answer: *directory*

3. Data layout on disk

2 driving forces

1) technology: avoid seeks, rotation

(last time)

2) workloads:

How do users access files?

1. Sequential access – bytes read in order (give me the next X bytes, then give me the next)
2. Random access - read/write elements out of middle of array (give me bytes j-k)

How are files typically used?

1. Most files are small (e.g. .login, .c files)
2. Large files use up most of the disk space
3. Large files account for most of the bytes transferred to/from disk

Bad news: need everything to be efficient

- Need small files to be efficient since lots of them
- need large files to be efficient, b/c most of the disk space, most of the I/O due to them

4. header -> blocks

How do we organize files on disk?

recall – seeks are slow,

for good bandwidth lay data out on disk sequentially

2 tasks

- (1) find i th block of a file easily
- (2) quickly access i th block of file

common data structures

file header – one per file; which disk sectors are associated with each file

- Head of linked list, array, root of tree → find i th block of file

What about performance

Separate mechanism from policy – once I can find where i th block of file is no matter where it is, then I have freedom to place any block anywhere → policy choice to lay data out sequentially when possible.

TO support such policies:

free space (bitmap) – 1 bit per block or sector; blocks numbered in cylinder-major order, so that adjacent numbered blocks can be accessed without seeks or rotational delay

Other aspect of performance

caching – every OS today keeps a cache of recently used disks blocks in memory to avoid having to go to disk. Common to all organizations. For now, assume no cache; add it later.

4.1 contiguous allocation

User says in advance how big file will be

Search bit map (using best fit/first fit) to locate space for file

File header contains:

- first sector in file
- file size (# sectors)

Pros & cons:

- + fast sequential access
- + easy random access

DA: external fragmentation
 DA: hard to grow files

4.2 Linked files

Each block, pointer to next on disk (Xerox Alto)

(DRAW PICTURE)

file header – points to first block on disk

Pros&cons

+ can grow files dynamically

+ free list managed same as file

DA: sequential access horrible: seek between each block

DA: random access is horrible²

DA: unreliable (lose block, lose rest of file)

4.3 FAT (MS-DOS, Windows9x, OS2)

Store linked list in separate table ("File allocation table")

A table entry for each block on disk

Each table entry in a file has pointer to next table entry in file (with special "eof" value to mark end)

Use "0" value to mean "free" (why not just put free elements on linked free list?)

compare to linked allocation

Sequential access

OK if FAT is cached

How much memory to cache entire FAT?

10GB disk/1KB sector = 10M entries ~40MB!

1TB disk/1kb sector → 1B entries ~8GB

→ FAT allocates larger "clusters"

→ policy -- try to allocate different parts of file near each other
 (reduces disk seeks and improves FAT cachability)

Random access

Reliability

Can replicate FAT if you want...

4.4 Indexed files (VMS)

User declares max file size

file header holds array of pointers big enough to point to file size number of blocks

<PICTURE>

- + can easily grow up to space allocated for descriptor
- + random access is fast
- DA: clumsy to grow file bigger than table size
- DA: still lots of seeks

4.5 Multilevel index (Unix 4.1)

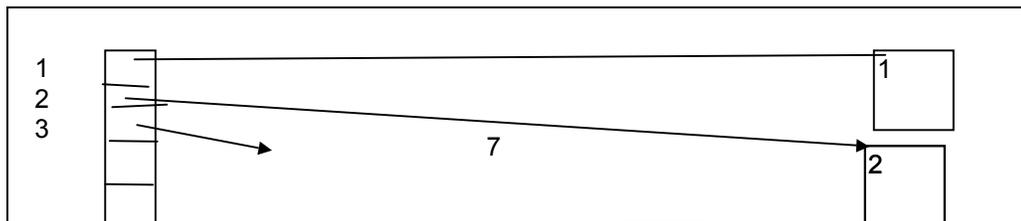
Key idea: efficient for small files, but still allow big files

file header "inode" -- per-file data structure
 -- array of inodes in fixed location of disk
 -- given *inumber* can find inode

inode = metadata (owner, creation time, isDirectory, size, ...)

13 pointers (fixed sized table; *not all pointers equivalent*)

- first 10 – point to data blocks
- let's assume 1KB blocks (can be any number of sectors)
- 11th – pointer to indirect block - pointer to a block of pointers
 - gives us 256 blocks + 10 from file header = 1/4 MB
- what if you allocate a 267th block? Pointer to doubly indirect block – a block of pointers to indirect blocks (in turn block of pointers to data blocks); gives about 64K blocks → 64MB
- triply-indirect block – block of pointers to doubly indirect blocks (which are...); gives about 16M blocks --> 16GB



- 1) Bad news: still an upper limit on file size (16 GB)
- 2) pointers get filled in dynamically: need to allocate indirect blocks only when file grows > 10 blocks; if small file, no indirection needed
- 3) How many disk accesses to reach block #23? Which are they?
 Block 5?
 Block 340?

UNIX pros&cons

+ simple (more or less)

+ files can easily expand (up to a point)

+ small files particularly cheap and easy

DA: very large files spend lots of time reading indirect blocks
 (but caching makes sequential I/O work well)

DA: lots of seeks

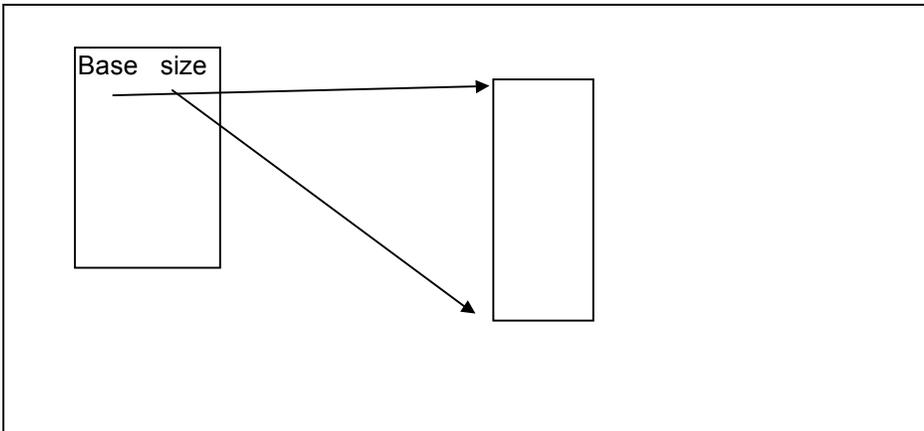
4.6 DEMOS

OS for Cray1 (mid to late 70's) – File system approach corresponds to segmentation

Cray1 had 12ns cycle time, so CPU:disk speed ratio about the same as today (a few million instructions = 1 seek)

Idea: reduce disk seeks by using contiguous allocation in normal case, but allow flexibility to have non-contiguous allocation

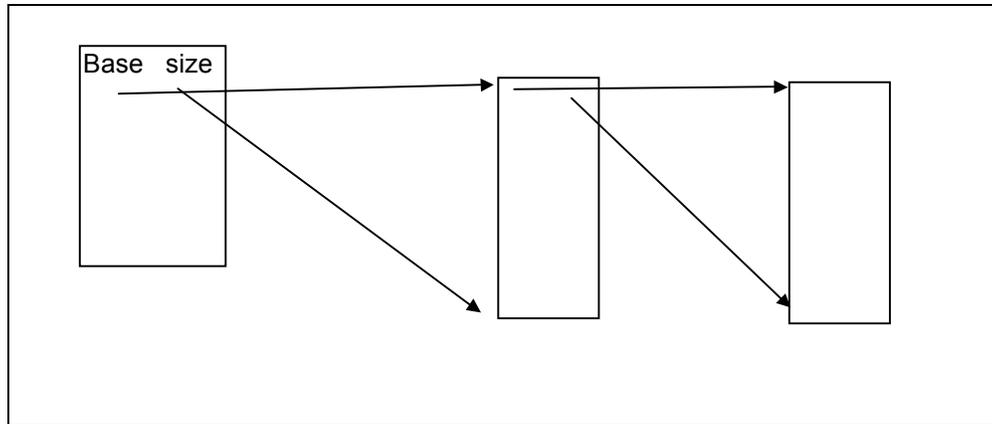
file header table of base&size (10 “block group” pointers)



Each “block group” a contiguous region of blocks

Are 10 block groups pointers enough? No. If need more than 10 block groups, set flag in file header BIGFILE

→ each table entry now points to an indirect block group – a block group of pointers to block groups



Can get huge files this way: suppose 100 blocks in a block group (can be bigger or smaller) → 10GB file size

QUESTION: How do you allocate a block group?

A: use bit map to find block of 0's

Pros&cons

- + easy to find free block group
- + free areas merge automatically

DA when disk fills up

- a) no long runs of free blocks (fragmentation)
- b) high CPU overhead to find free block

In practice disks are always full!

Solution: don't let disks get full – keep pointers in reserve
normally, don't even try to allocate if free count == 0
change this to

don't allocate if free count < reserve

Why do this?

Tradeoff – pay for more disk space, get contiguous allocation

How much reserve do you need?

In practice, 10% seems like enough

4.7 UNIX BSD 4.2

(Most current unices)

Policy v. mechanism

Same mechanisms as BSD 4.1 (same file header, triply indirect blocks)

except incorporate some ideas from DEMOS:

- uses bitmap allocation in place of free list
- attempt to allocate files contiguously
- 10% reserve disk space
- skip sector positioning

Problem: when you create a file, don't know how big it will become (in UNIX most writes are by appending to file) So how much contiguous space do you allocate for a file when it is created?

In Demos, power of 2 growth: once it grows past 1MB, allocate 2MB, etc

IN BSD 4.2, just find some range of free blocks, put each new file at front of a different range. When need to expand a file, you first try successive blocks in bitmap

sequential files

processing overhead can cause problems

read 1 block, process, read next block...

While processing, disk was turning. If have to wait for entire rotation, bad performance. GO from reading @ disk BW to reading 1 sector per rotation

Two solutions:

- skip sector positioning (BSD 4.2)

- read ahead/disk track buffers – read next block right after first, even if application hasn't asked for it yet. This could be done either by OS (read ahead) or disk itself (track buffers)

4.8 NTFS

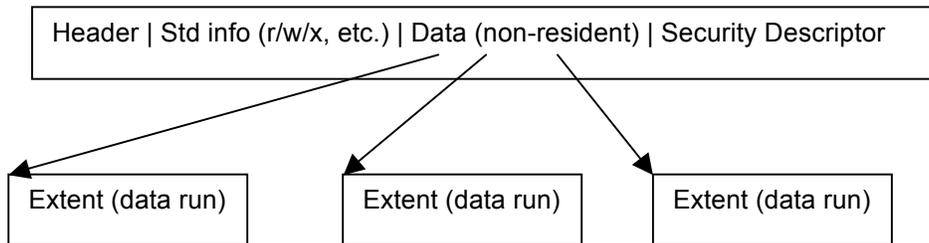
Master file table – table of MFT entries (MFT entry = file header, typically 1-4 KB)

Data can be “resident” in header (for small file), or header can have pointers to data extents for larger file (array of <**file offset, disk offset, length**>), or header can have pointer to another MFT entry that holds nothing but pointers to extents (for large files), or header can have pointer to several MFT entries that hold nothing but pointers to extents (for really huge fragmented files)

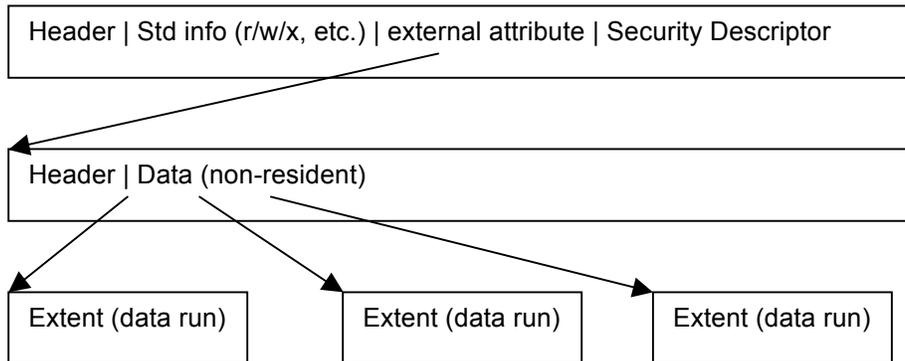
Small file:



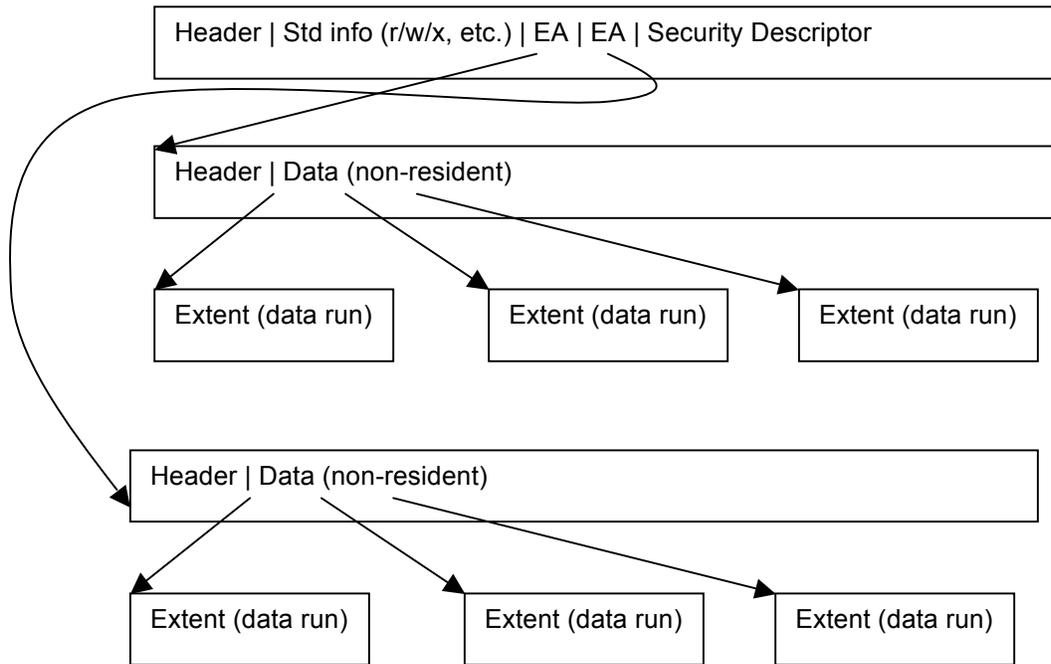
Large file:



Huge file:



Extremely huge file:



5. Policy v. mechanism

Separating mechanism from policy can be useful

What properties should a file index mechanism have in order to support the widest range of policies?

Which of the protocols listed above (contiguous, linked, FAT, index, multi-level index, demos) have sufficient mechanism to allow flexible policy?

Admin - 3 min

Drafts of projects 2-3 available

Lecture - 23 min

Summary - 1 min
