Lecture 17: File Systems and Disk Management

17.0 Main Points

Implementing file system abstraction

Comparison among disk management algorithms

Physical Reality	File System Abstraction
block oriented	byte oriented
physical sector #'s	named files
no protection	users protected from each other
data might be corrupted if machine crashes	robust to machine failures

17.1 File System Components

Disk management: how to arrange collection of disk blocks into files

Naming: user gives file name, not track 50, platter 5, etc.

Protection: keep information secure

Reliability/durability: when system crashes, lose stuff in memory, but want files to be durable.

17.2 User vs. System View of a File

User's view:

durable data structures.

Systems's view (system call interface):

collection of bytes (UNIX)

System's view (inside OS):

collection of blocks (a **block** is a logical transfer unit, while a sector is the physical transfer unit. Block size multiple of the sector size; in UNIX, block size is 4KB.)

17.2.1 Translating from user to system view

What happens if user says: give me bytes 2 - 12?

fetch block corresponding to those bytes return just the correct portion of the block

What about: write bytes 2 - 12?
fetch block
modify portion
write out block

17.3 File Usage Patterns

How do users access files?

1. Sequential access -- bytes read in order (give me the next X bytes, then give me next)

2. Random access -- read/write element out of middle of array
(give me bytes i-j)

3. Content-based access -- "find me 100 bytes starting with ANDERSON"

Most file systems don't provide #3; instead, database is built on top to index content (requires efficient random access)

How are files typically used?

1. Most files are small (for example, .login, .c files)
2. Large files use up most of the disk space

3. Large files account for most of the bytes transfered 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, since most of the disk space, most of the I/O due to them

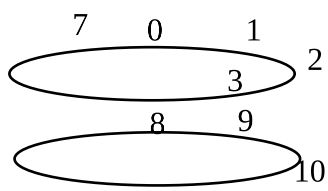
17.4 Disk Management Policies

How do we organize file on disk?

17.4.1 Common data structures

Need a "file header", one for each file: which disk sectors are associated with each file

Also, need bitmap to represent free space on disk, one bit per block (or sector). Blocks numbered in cylinder-major order, so that adjacent numbered blocks can be accessed without seeks or rotational delay.



Track 0, surface 0, sector 0, 1, ... | surface 1, sector 0, 1 ... | ... | track 1, surface 1, sector 0...

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

17.4.2 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 (# of sectors)

Pros & cons:

+ fast sequential access

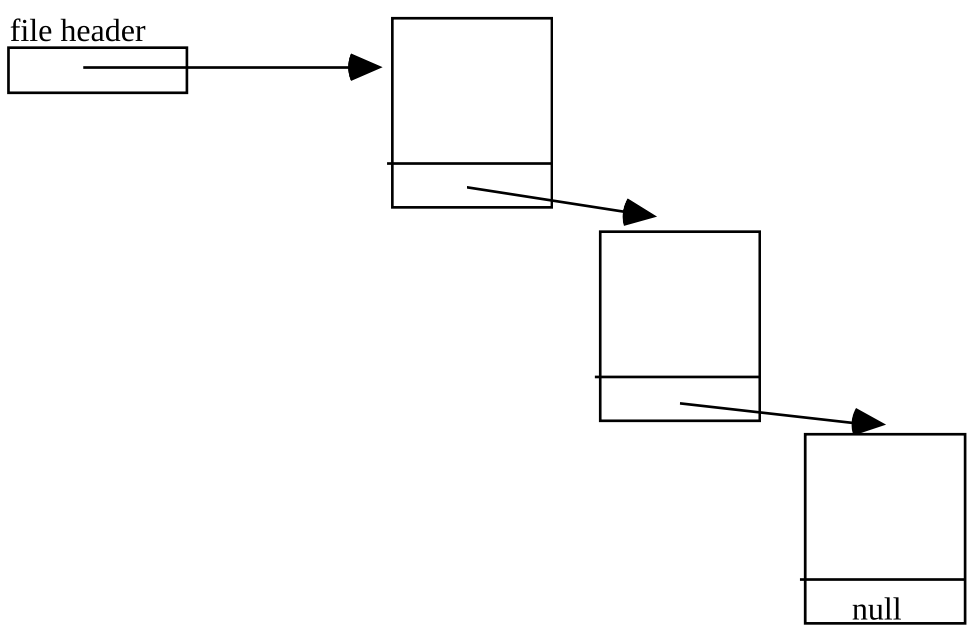
+ easy random access

- external fragmentation

- hard to grow files

17.4.3. Linked files

Each block, pointer to next on disk (Alto)



File header pointer to first block on disk

Pros&cons:

+ can grow files dynamically

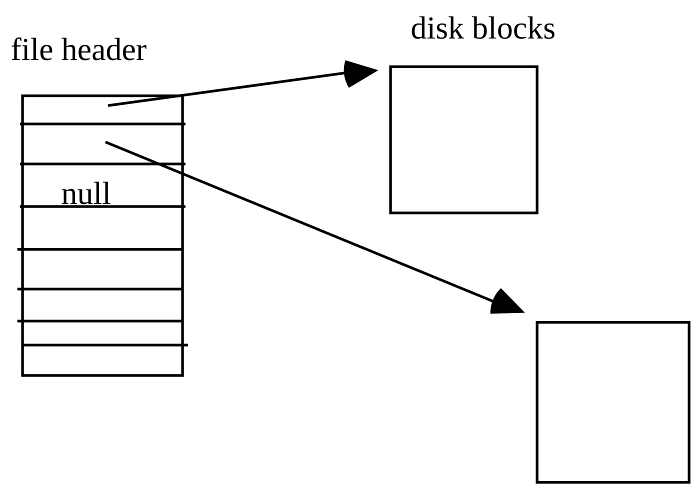
- sequential access: seek between each block

- - - random access: horrible

- unreliable (lose block, lose rest of file)

17.4.3 Indexed files (Nachos, VMS)

User declares max file size; system allocates a file header to hold an array of pointers big enough to point to file size number of blocks.



+ can grow up to space allocated for descriptor

+ random access is fast
- clumsy to grow file bigger than table size

- still lots of seeks

17.4.5 Multilevel indexed (UNIX 4.1)

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

File header: 13 pointers (fixed size table, pointers not all equivalent)

First ten, pointers to data blocks. (If file is small enough, some pointers will be NULL.)

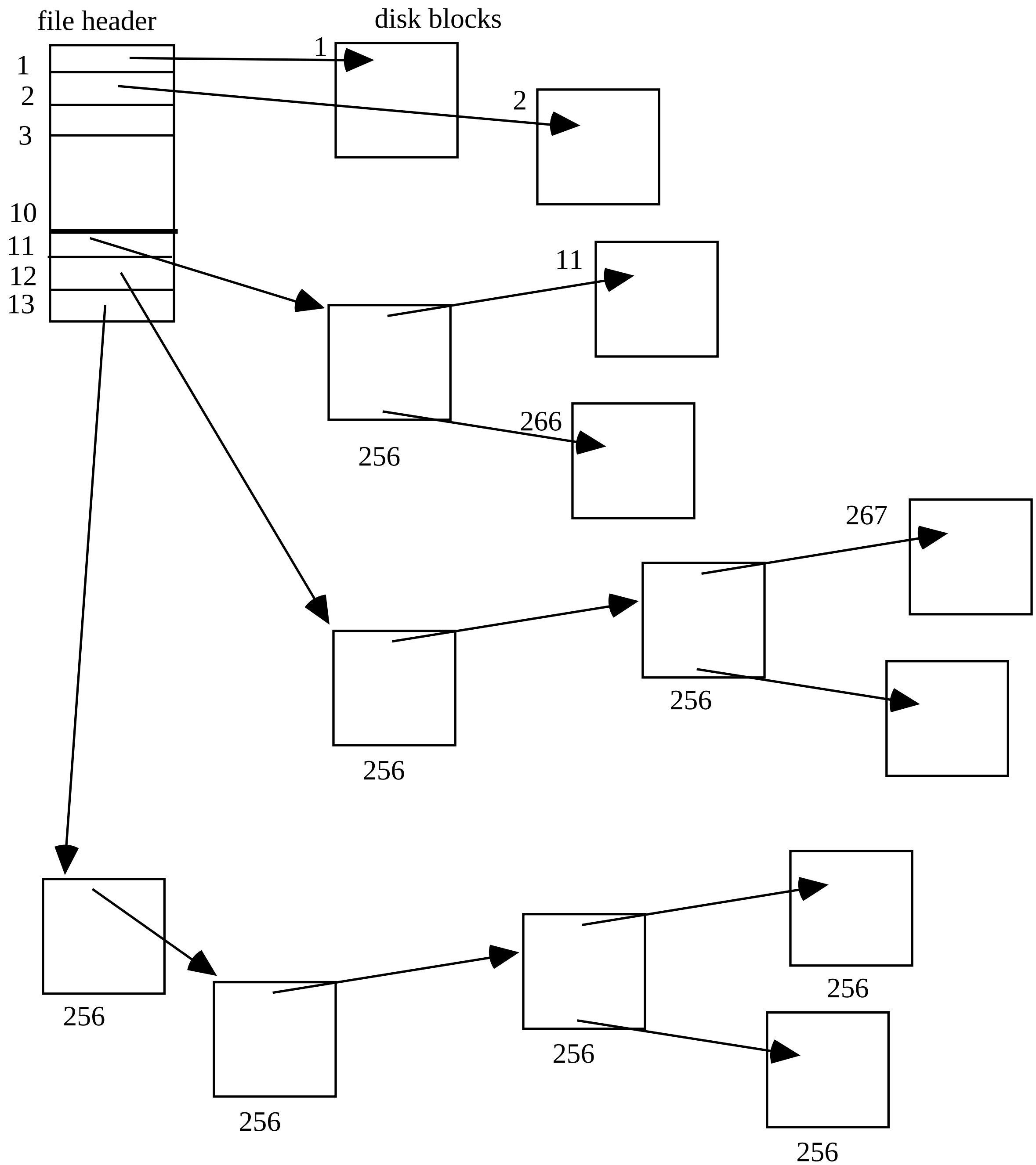
What if you allocate 11th block?
Pointer to an indirect block -- a block of pointers to data blocks. Gives us 256 blocks, + 10 (from file header) = 1/4 MB

What if you allocate a 267th block?
Pointer to a doubly indirect block -- a block of pointers to indirect blocks (in turn, block of pointers to data blocks). Gives us about 64K blocks => 64MB

What if want a file bigger than this? One last pointer -- what should it point to?

Instead, pointer to triply indirect block -- block of pointers to doubly indirect blocks (which are ...)

Thus, file header is:
First 10 data block pointers
 11 indirect block pointer
 12 doubly indirect block pointer
 13 triply indirect block pointer



1. Bad news: Still an upper limit on file size ~ 16 GB.
2. Pointers get filled in dynamically: need to allocate indirect block only when file grows > 10 blocks. If small file, no indirection needed.

3. How many disk accesses to reach block #23? Which are they?

How about block # 5?

How about block # 340?

UNIX pros&cons:

+ simple (more or less)

+ files can easily expand (up to a point)

+ small files particularly cheap and easy

- very large files, spend lots of time reading indirect blocks

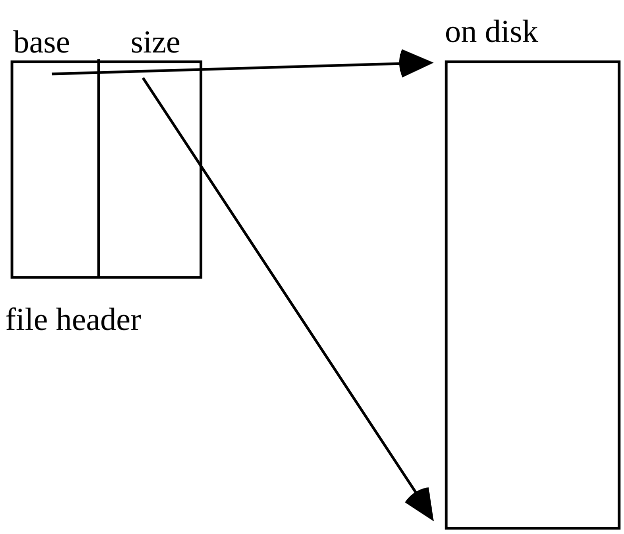
- lots of seeks

17.4.6 DEMOS

OS for Cray-1, mid to late 70's. Cray-1 had12 ns 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.

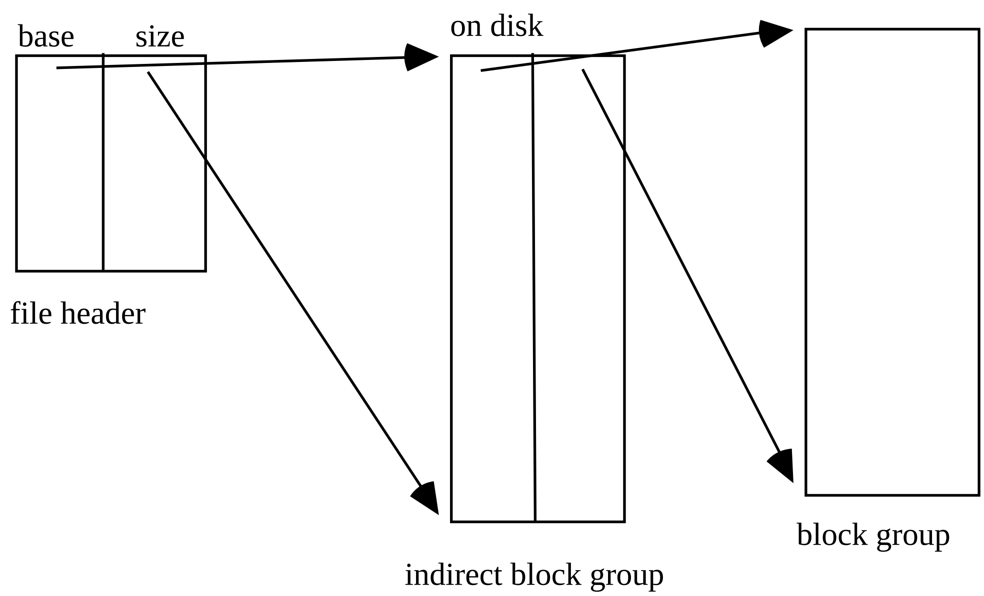
File header: table of base&size (10 "block group" pointers)



Each "block group" -- a contiguous region of blocks

Are10 block group 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) => 10 GB max file size

How do you find a block group? Use bit map to find block of 0's

Pros&cons:

+ easy to find free block groups

+ free areas merge automatically

- when disk fills up:

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

In practice, disks are always full.

Solution:
don't let disk get full -- keep portion in reserve

Free count = # blocks free in bitmap.
Normally, don't even try 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 of a reserve do you need?
In practice, 10% seems like enough.

17.4.7 UNIX BSD 4.2

Exactly the same as BSD 4.1 (same file header and triply indirect blocks), except incorporated some ideas from DEMOS:
uses bitmap allocation in place of free list
attempt to allocate files contiguously
10% reserved disk space
skip sector positioning

Problem: how do you know how big a chunk of space to allocate when a file is created?

In Demos, power of 2 growth: once it grows past1 MB, allocate 2MB, etc.

In BSD 4.2, put each new file at the front of a different range of free blocks.
When need to expand a file, find free block closest to previous blocks in file.

Problem: what about rotational delay? With sequential file, common case is read one block, do processing, read next block. In the meantime, disk has continued turning.

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 by disk itself (track buffers).

17.5 Disk scheduling

Disk can do only one request at a time; what order do you choose to do requests?

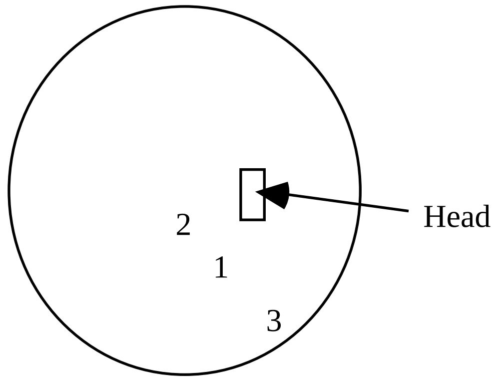
If 0 or 1 request queued, easy. But what if more than 1? Try to arrange requests in some order that reduces seek time.

17.5.1 FIFO order

Fair among requesters, but order of arrival may be to random spots on the disk => long seeks

17.5.2 SSTF

SSTF: shortest seek time first. Pick the request that's closest on the disk. (Although called SSTF, today, include rotational delay in calculation, since rotation can be as long as seek.)



Order requests will be serviced using SSTF.

SSTF is good at reducing seeks, but may get starvation

17.5.3 SCAN

SCAN implements an elevator algorithm: take the closest request in the direction of travel. No starvation, but retains flavor of SSTF.