Lecture# 20: File system – data layout, naming

****
Review -- 1 min
****

Intro to I/O – overhead, latency, BW
Disks – avoid seeks, rotations
Header – given file header find rest of file
- Contiguous allocation
- Linked allocation
- FAT
- Indexed allocation
- Multi-level index

****
Outline - 1 min
********** *****

Data layout
- files: file header, locating the rest of the file

Contiguous allocation
- Linked allocation
- FAT
- Indexed allocation
- Multi-level index

- naming, directories – given name, find header

****
Preview - 1 min
****

Wednesday: Guest lecture – IBM JFS and volume mgr
Next time: finish file systems
- In-memory data structures
- Scheduling
- Transactions

****
Lecture - 35 min
****

1.1 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

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

1.3 FAT (MS-DOS, Windows9x, OS2)

Store liked 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?
40GB disk/1KB sector = 40M entries ~~160MB!
→ FAT allocates larger "clusters"
→ policy – try to allocate different parts of file near each other (reduces disk seeks and improves FAT cachability)

Random access
OK if FAT cached

Reliability
Can replicate FAT if you want…(need to keep copies in sync…)

1.4 Indexed files (VMS)

User declares max file size

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

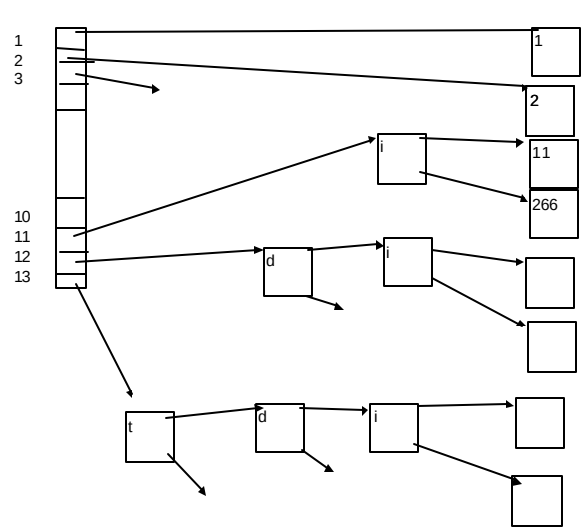
<PICTURE>

+ 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

1.5 Multilevel index (Unix 4.1)

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

file header 13 pointers (fixed sized table; *not all pointers equivalent*)
- first 10 point to data blocks
- 11th – pointer to indirect block - pointer to a block of pointers
  - gives us 256 blocks + 10 from file header = ¼ 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…)



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

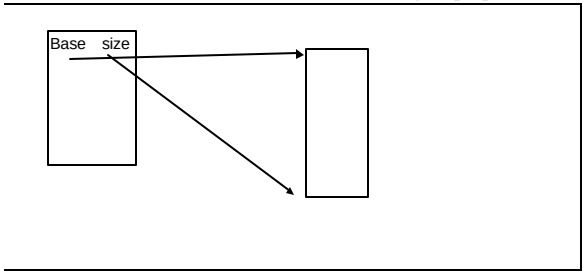
1.6 DEMOS

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)

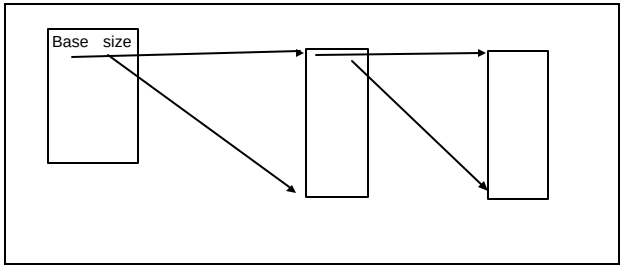
Idea: reduce disk seeks by using contigous allocation in normal case, but allow flexibility to have non-contiguous allocation

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 CPUoverhead to find free block

In practice disks are always full!

Solution: don't let disks get full – keep pointers in reserve
   normally, don't 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

1.7 UNIX BSD 4.2 – FFS fast file system

(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, allocte 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. Start files from same directory near each other

2. 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
****
Project 2-3 available
Guest lecture Wednesday – 5-6:30PM

****

3. Naming and directories

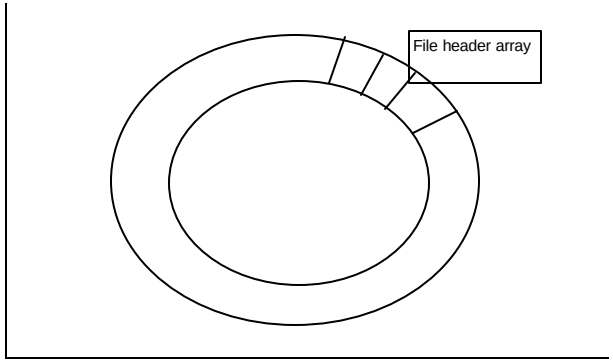
ok: once I find a file header, I can find the rest of the file
How do you find a file header? **Name lookup**

Directory: table of
<name> <pointer to file header>

3.1 File header

Where is file header stored on disk?

In (early) unix – special array in outermost cylinders:



Unix refers to file by index into array – tells it where to find the file header
UNIX-isms
"I-node" – file header
"I-number" – index into array
i-number is unique identifier for file

Unix file header organization seems strange
1) header not anywhere near data blocks. To read a small file, seek to get header, seek back to data
2) Fixed size, set when disk formatted. Means maximum # files that can be created

Why not put headers near data?
+ reliability – whatever happens to the disk, you can find all of the files
+ Unix BSD 4.2 puts portion of the file header array on each cylinder. For small directories – can fit all data, file headers, etc in same cylinder – no seeks!
+ file headers are much smaller than a whole block (a few hundred bytes), so multiple file headers fetched from disk at same time

QUESTION: do you ever look at a file header w/o reading the file?
If not – put the file header in first block of the file!

Turns out – fetching the file header is about 4x more common in Unix than reading the file (ls, make, etc)

Bottom line:
array of headers (array of inodes)
index of array ("inumber") → header → bytes of a file

3.2 Naming

3.2.1 Options

1. Use index (ask user to specify I-node number)
2. text name
3. icon

With icon or text, need to map name→index

3.2.2 Directories

Directory maps name→ file index (where to find file header)

Directory just a table of file name, file index pairs
I could write it on a piece of paper and carry it around in my pocket…

Directory is just a file
Only OS permitted to modify directory
- so it always contains name→ file index

Any program can read directory file
this is how ls works

3.2.3 Directory hierarchy

Take this one step at a time, starting at the bottom:

Ok – how do you find blocks of a file?
→ find its header – header has pointers to blocks of a file
how do you find a header?
→ find its I-number – inumber is pointer to header
how do you find a file's inumber?
→ read its directory – directory maps name to inumber
But wait, directory is a file – how do you find *it*?

How do you find a file?
→ find its header
…

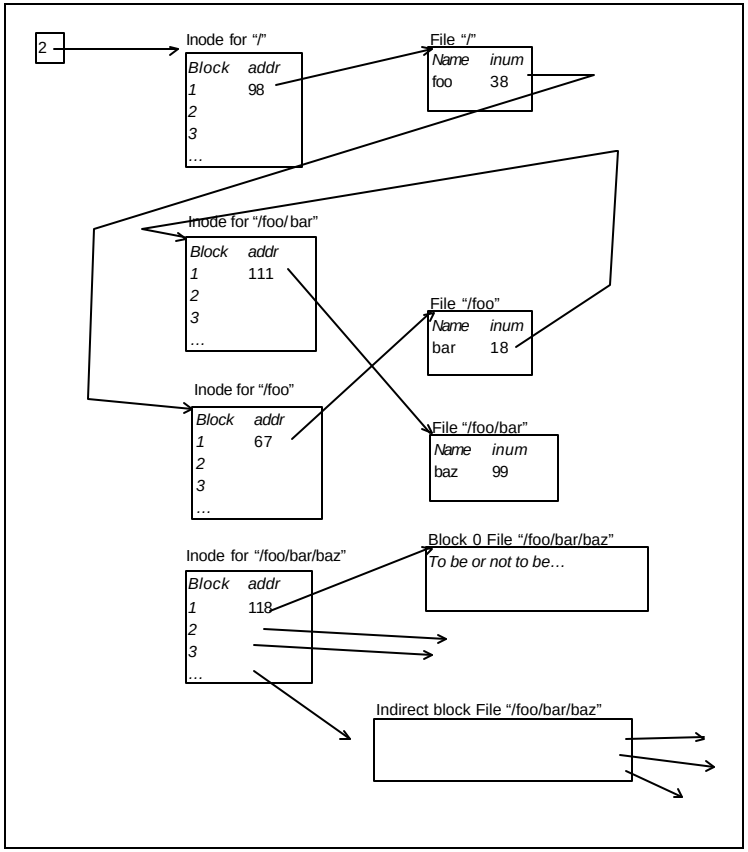
You're seniors – a little recursion shouldn't bother you!
→ lets you nest directories – directories of directories, etc

Interpret:
/foo/bar/baz

- baz is a file
- bar/ is a directory that contains the inumber of file baz
- foo/ is a directory that contains the inumber of file bar

What do you need with recursion? A base case

- "/" is a directory that contains the inumber of file foo
- the inumber of "/" is 2



How many disk I/Os needed to access first byte of file

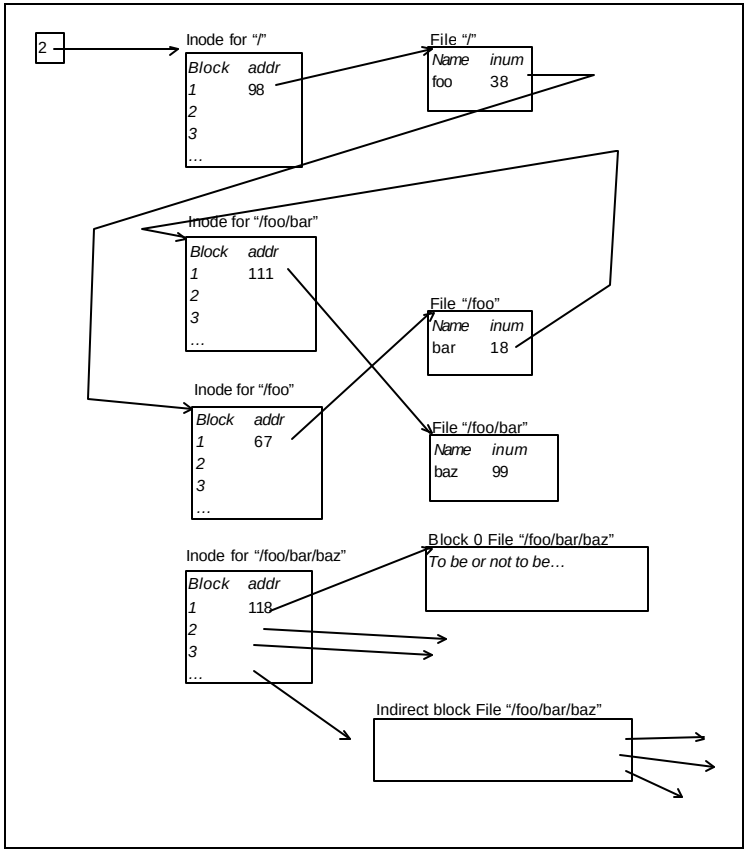
1) read in file header for root (always from fixed location)
2) read in first data block for root
3) read in file header for foo
4) read in first block of foo

5) read in file header for bar
6) read in first block of bar
7) read in file header for baz
8) read in first data block for baz

current working directory: short cut for both user and system.
Each address space stores file index for current directory.
Allows user to specify relative file name instead of absolute path (if no leading "/")

Thus, to read first byte of file, just last 4 steps above

How can this possibly be efficient? Caching!



How many disk I/Os needed to access firs byte of file
1) read in file header for root (always from fixed location)
2) read in first data block for root
3) read in file header for foo
4) read in first block of foo

5) read in file header for bar
6) read in first block of bar
7) read in file header for baz
8) read in first data block for baz

current working directory: short cut for both user and systm.
Each address space stores file index for current directory.
Allows user to specify relative file name instead of absolute path (if no leading "/")

Thus, to read first byte of file, just last 4 steps above

How can this possibly be efficient? Caching!

****
Summary - 1 min
****