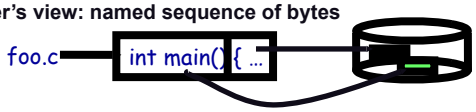
CS 140 - Summer 2008 - Handout #16: File Systems (pt. 1)

File Systems: Making Disks Useful

- Next: bottom up exposition of file systems
- Files:
  - **What they look like**
  - **Why**
  - **Some implications**
- We ignore many other, richer storage system organizations (e.g., databases)

Files: named bytes on disk

- File abstraction:
  - **User's view: named sequence of bytes**

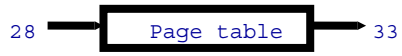


  - **FS's view: collection of disk blocks.**
  - **File system's job: translate name & offset to disk blocks.**

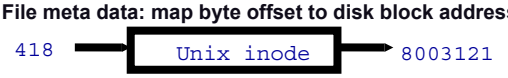
- File operations:
  - **Create a file, delete a file.**
  - **Read from file, write to file.**
- Want: operations to have as few disk accesses as possible & have minimal space overhead

What's so hard about grouping blocks???

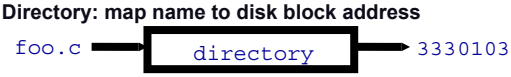
- In some sense, the problems we will look at are no different than those in virtual memory
  - **Like page tables, file system meta data are simply data structures used to construct mappings.**
  - **Page table: map virtual page # to physical page #**



  - **File meta data: map byte offset to disk block address**



  - **Directory: map name to disk block address**



FS vs.VM

- In some ways problem similar:
  - **want location transparency, oblivious to size, & protection.**
- In some ways the problem is easier:
  - **CPU time to do FS mappings not a big deal (= no TLB).**
  - **Page tables deal with sparse address spaces and random access, files are dense (0 .. filesize-1) & ~sequential.**
- In some ways problem is harder:
  - **Each layer of translation = potential disk access.**
  - **Disk is huge, but... cache space never enough, the amount of data you can get with one fetch never enough.**
  - **Range very extreme: Many <10k, some more than GB.**
  - **Implications?**

Some working intuitions

- FS performance dominated by # of disk accesses
  - **Each access costs 10s of milliseconds.**
  - **Touch the disk 50-100 extra times = 1 *second*.**
  - **Can easily do 100s of millions of ALU ops in same time.**
- Access cost dominated by movement, not transfer
  - **Can get 20x the data for only ~5% more overhead.**
  - **1 sector = 10ms + 8ms + 50us (512/10MB/s) = 18ms.**
  - **20 sectors = 10ms + 8ms + 1ms = 19ms.**
- Observations:
  - **All blocks in file tend to be used together, sequentially.**
  - **All files in a directory tend to be used together.**
  - **All names in a directory tend to be used together.**
  - **How to exploit?**

Common addressing patterns

- Sequential:
  - **File data processed in sequential order.**
  - **By far the most common mode.**
  - **Example: editor writes out new file, compiler reads in file, etc.**
- Random access:
  - **Address any block in file directly without passing through predecessors.**
  - **Examples: data set for demand paging, databases.**
- Keyed access:
  - **Search for block with particular values.**
  - **Examples: associative data base, index.**
  - **Usually not provided by OS.**

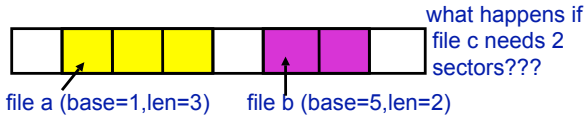
Problem: how to track file's data?

- Disk management:
  - **Need to keep track of where file contents are on disk.**
  - **Must be able to use this to map byte offset to disk block.**
  - **The data structure used to track a file's sectors is called a file descriptor.**
  - **file descriptors often stored on disk along with file.**
- Things to keep in mind while designing file structure:
  - **Most files are small.**
  - **Much of the disk is allocated to large files.**
  - **Many of the I/O operations are made to large files.**

  Want good sequential and good random access (what do these require?)

Simple mechanism: contiguous allocation

- "Extent-based": allocate files like segmented memory
  - **When creating a file, make the user specify pre-specify its length and allocate all space at once.**
  - **File descriptor contents: location and size.**

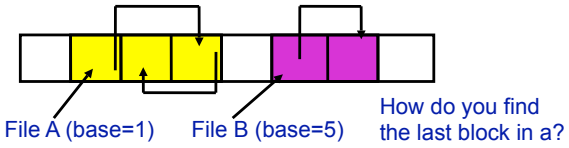


Example: IBM OS/360.

Pro: simple, fast access, both sequential and random.
Cons? (What does VM scheme does this correspond to?)

Linked files

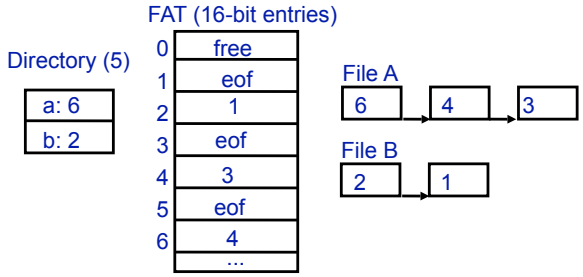
- Basically a linked list on disk.
  - **Keep a linked list of all free blocks.**
  - **File descriptor contents: a pointer to file's first block in each block, keep a pointer to the next one.**



Pro: easy dynamic growth & sequential access, no fragmentation
Con?
Examples (sort-of): Alto, TOPS-10, DOS FAT

Example: DOS FS (simplified)

- Uses linked files. Cute: links reside in fixed-sized "file allocation table" (FAT) rather than in the blocks.



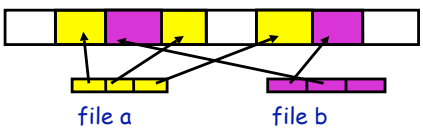
Still do pointer chasing, but can cache entire FAT so can be cheap compared to disk access.

FAT discussion

- Entry size = 16 bits
  - **What's the maximum size of the FAT?**
  - **Given a 512 byte block, what's the maximum size of FS?**
  - **One attack: go to bigger blocks. Pro? Con?**
- Space overhead of FAT is trivial:
  - **2 bytes / 512 byte block = ~.4% (Compare to Unix)**
- Reliability: how to protect against errors?
  - **Create duplicate copies of FAT on disk.**
  - **State duplication a very common theme in reliability.**
- Bootstrapping: where is root directory?
  - **Fixed location on disk:**

Indexed files

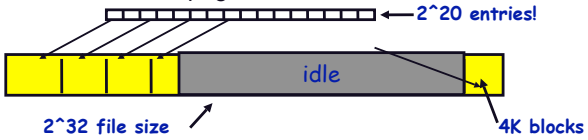
- Each file has an array holding all of its block pointers
  - **(purpose and issues = those of a page table)**
  - **Max file size fixed by array's size (static or dynamic?)**
  - **Create: allocate array to hold all file's blocks, but allocate on demand using free list.**



Pro: both sequential and random access easy
con?

Indexed files

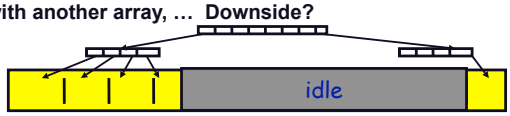
- Issues same as in page tables



Large possible file size = lots of unused entries

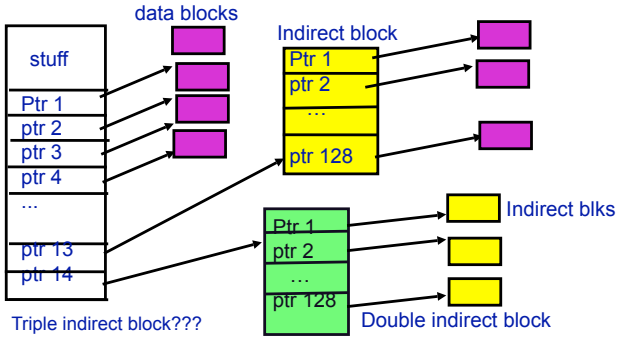
Large actual size? table needs large contiguous disk chunk

Solve identically: small regions with index array, this array with another array, … Downside?



Multi-level indexed files: ~4.3 BSD

- File descriptor (inode) = 14 block pointers + "stuff"



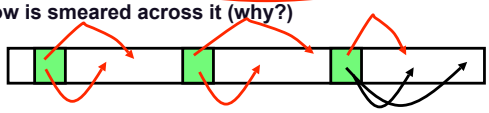
Unix discussion

- Pro?
  - **Simple, easy to build, fast access to small files.**
  - **Maximum file length fixed, but large. (With 4k blks?)**
- Cons:
  - **What's the worst case # of accesses?**
  - **What's some bad space overheads?**
- An empirical problem:
  - **Because you allocate blocks by taking them off unordered freelist, meta data and data get strewn across disk**

More about inodes

- Inodes are stored in a fixed sized array
  - **Size of array determined when disk is initialized and can't be changed. Array lives in known location on disk. Originally at one side of disk:**

Now is smeared across it (why?)



The index of an inode in the inode array called an i-number. Internally, the OS refers to files by inumber

When file is opened, the inode brought in memory, when closed, it is flushed back to disk.