Disk Allocation

CS 537 - Introduction to Operating Systems

Free Space

- Need to keep track of which blocks on a disk are free
- Disk space is allocated by sectors
  - 512 bytes typically
- The list of free blocks is also stored on disk

Bit Vector

- A very simple method is to keep a single bit for each block on disk
- Example
  - Free blocks: 2, 5, 13, 14, 15, 23, 24, 29, 31, …
  - Bit Vector: 00100100000001110000000110000101…
- Requires the use of some disk space
  - a 16 GB disk would require 8192 blocks to map free list (assuming 512 byte blocks)
  - roughly 0.025% of entire disk space

Bit Vector

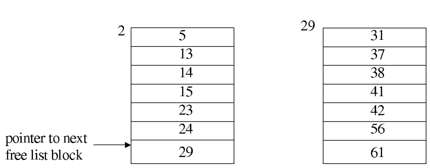
- Fairly simple to implement
  - requires hardware support for bit manipulation
- Biggest advantage is the ability to select whichever block
  - can be used to pick adjacent blocks for a file
- For good performance, cache the bit vector in memory
  - allows for fast lookup of available blocks
  - need to write the vector back to disk frequently
    - crash recovery

Linked List

- Keep a pointer in each free block to the next free block
- To find a free block, just grab the first block off the list
- Problem arises if multiple free blocks are needed
  - have to follow links all over disk - poor performance

Grouped Linked List

- A single free block will point to a group of other free blocks
- Consider the following free blocks
  - 2, 5, 13, 14, 15, 23, 24, 29, 31, 37, 38, 41, …



Grouped Linked List

- The last entry in each group points to another free block with pointers to more free blocks
- When all the blocks in a group have been allocated, then use the block that held the pointers
- Requires no disk space for implementing
  - just need to store the location of the first pointer block

Clusters

- Disk blocks are 512 bytes
- Most file systems group several blocks together to form a cluster
  - 1K, 4K, 16K, etc.
- Lowest levels of OS must deal with physical sectors
- Everything else can work on clusters
  - this includes the file system
  - think of them as logical sectors

Clusters

- 4 KB cluster fits nicely into a single page of memory
- This helps in prefetching data
- Internal fragmentation is now worse
  - not bad though if the average file is near 4 KB
  - or if most files are very large

Clusters

- Reconsider the bit vector requirements
  - 16 GB disk using 4KB clusters
  - each bit in the vector now represents 8 physical sectors
  - total memory requirements are now 1024 physical sectors

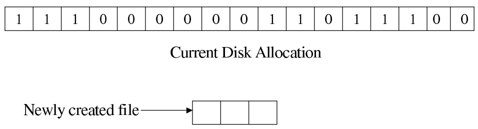
File Space Allocation

- Basic issues
  - most files change size over there life time
  - some files tend to be read sequentially
    - would like to allocate space sequentially on disk
  - some files are not read sequentially
    - database files for example
    - would still like to have decent performance
  - files are continuously created and deleted
    - this could cause fragmentation of disk
  - disks are slow
    - most information will be cached in memory

Contiguous Allocation

- When a file is first created, give it a set of contiguous blocks on disk
- Simple method to implement
  - just search free list for correct number of consecutive blocks and mark them as used
- Supports sequential access very well
  - files entire data is stored in adjacent blocks
- Also supports random access well
  - quick and easy to determine where any piece of data lives

Contiguous Allocation



This file could start in any of the blocks 3 through 6

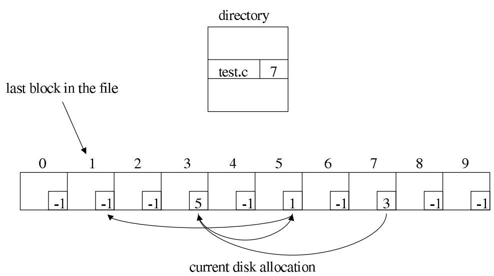
Contiguous Allocation

- Several big problems with this method
  - External fragmentation of disk
  - How to determine how much space a file should be given
    - default value? user defined?
  - What happens if file needs to grow beyond allocated space?
    - don't let it happen? copy it to a bigger space?

Linked Allocation

- Keep a pointer in each file block to the next block of the file
- Simple to imlement
  - directory just needs to keep track of the first block in the file
- Allows file to easily grow
- No external fragmentation

Linked Allocation



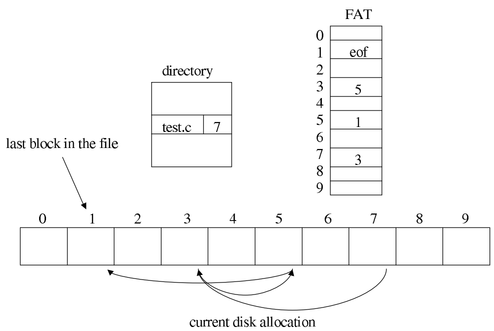
Linked Allocation

- A few problems with this method
  - a small portion of a files space is used for pointers instead of for data
    - not a huge issue
  - to find a random byte in the file, must search through all the other blocks to find the right pointer
    - poor for performance in non-sequential accesses
    - this is a huge issue

File Allocation Table (FAT)

- This is an extension of the linked allocation
- Instead of putting the pointers in the file, keep a table of the pointers around
- This table can be quickly searched to find any random block in the file

FAT



FAT

- All the blocks on disk must be included in the table
- Assume 4 KB clusters and 16 GB disk
  - number of entries in the FAT is about 4 million
  - assume each entry is 32 bits
  - size of the FAT is 128 MB
- A nice side effect of FAT is for the free list
  - whether a block is free or not can be recorded in the table

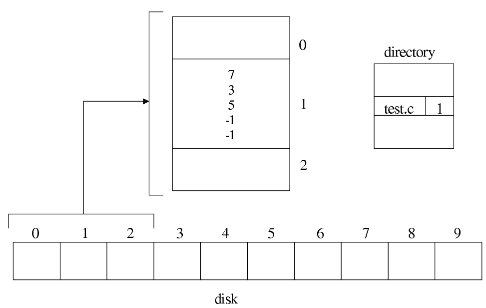
FAT

- For good performance, the FAT should be cached in memory
  - otherwise traversing the list would require many disk accesses to “follow” the pointers
- This method works well for both sequential and random access
- This is the method used by Windows

Indexed Allocation

- Another solution is to record all of the locations of a files blocks in a separate "file"
- This "file" is referred to as an index node
  - inode for short
- It contains all the pointers to the blocks that a file currently owns

Indexed Allocation



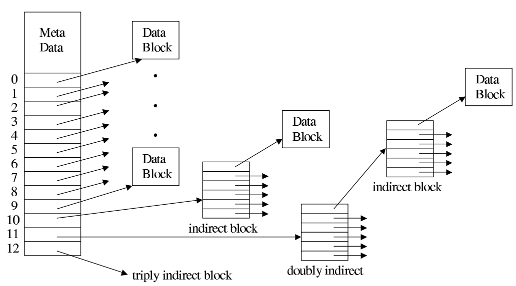
Indexed Allocation

- The amount of space a file can hold would be limited by the size of the inode
  - if only 10 entries fit in the inode, that would mean only 10 different blocks could be referenced
- To represent a large file, would need large inodes
- Large inode would be a waste of space for small files
- Use indirection!

Indexed Allocation

- Unix inodes are a total size of 128 bytes
  - the first part of an inode is the meta data for a file
  - a total of 13 pointers (each 4 bytes long)
- There are 10 *direct pointers*
  - point directly to data blocks for the file
- There is 1 *indirect pointer*
  - points to a block that contain only pointers to data blocks for the file
- There are also 1 *doubly indirect* and 1 *triply indirect* pointers

Indexed Allocation



Indexed Allocation

- Small files will use the direct pointers
  - little overall wasted space
- Large files will use the indirect pointers
  - allows for huge files
- Maximum number of pointers in an indirect block is 512/4 = 128 pointers
- Maximum file size (in blocks) is
  - $10 + 128 + 128*128 + 128*128*128 \approx 2$ million blocks

Indexed Allocation

- Where are the inodes stored?
  - in fixed location at beginning of disk
  - think of it as a big table of inodes
  - the root directory is stored at location 0 in the table

Indexed Allocation

- To read a single data block may require multiple accesses to disk
  - need to go through indirect pointers
- CACHE!
  - place a referenced inode and subsequent indirect pointer blocks into memory and access there