Files and Directories

Module 16

Filesystem - Implementor’s View

A file system provides a convenient set of illusions to the user - but it needs to do a lot of book-keeping to preserve the illusion!

Now we’ll study the data structures involved

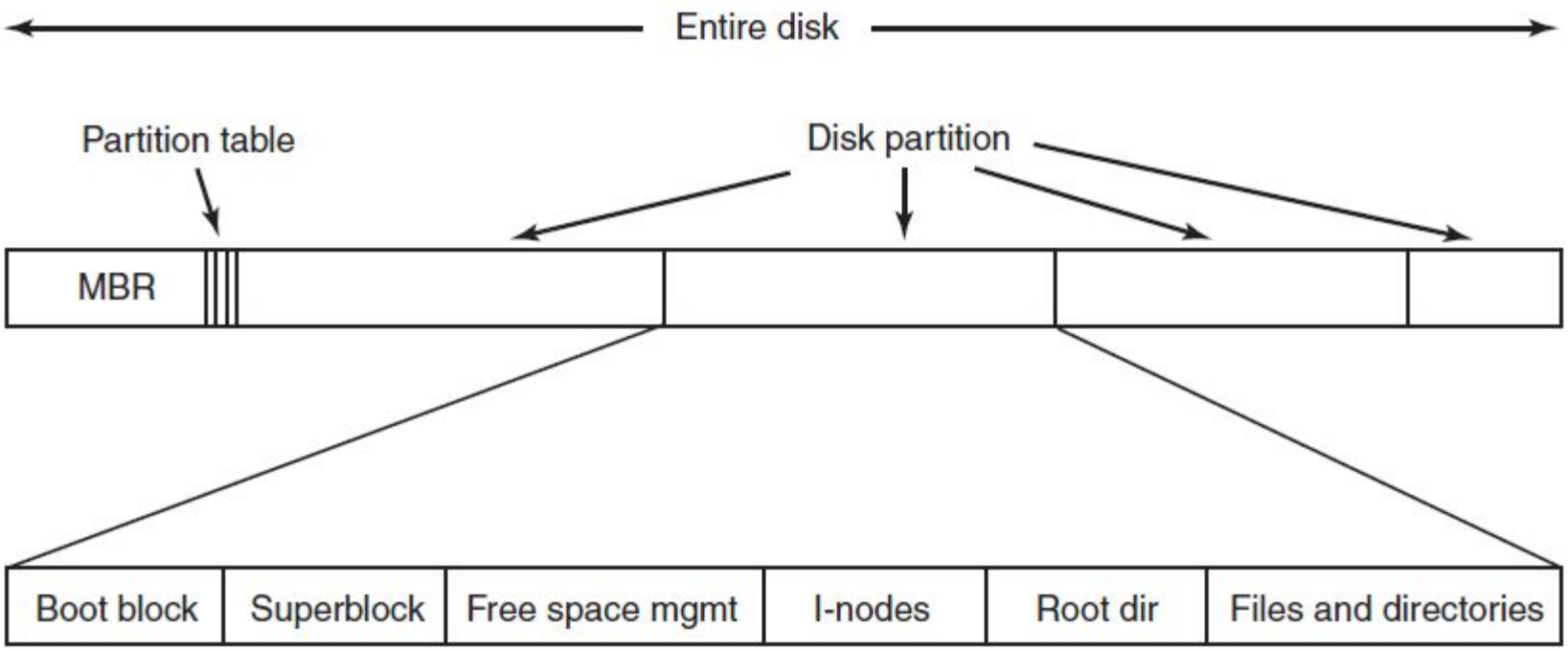
First off… how do we begin?

- File systems are stored on disks
- Disks can have several file systems - which occupy partitions.
- Some partitions may contain Operating Systems… but some won’t

First off… how do we begin?

- Each disc contains a Master Boot Record in sector/block 0 of the disc
- The MBR contains:
  - An executable program (execute by Firmware)
  - Partition Table
- The MBR executable will read the boot-blocks of each partition.
  - It may provide the user a choice of which OS to load, if multiple partitions indicate they have an OS in their boot-block.

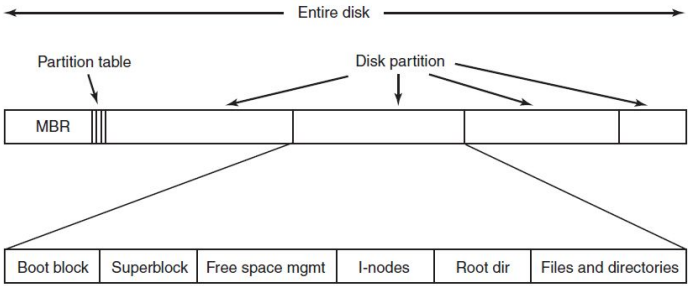
File System Layout



Beyond the MBR

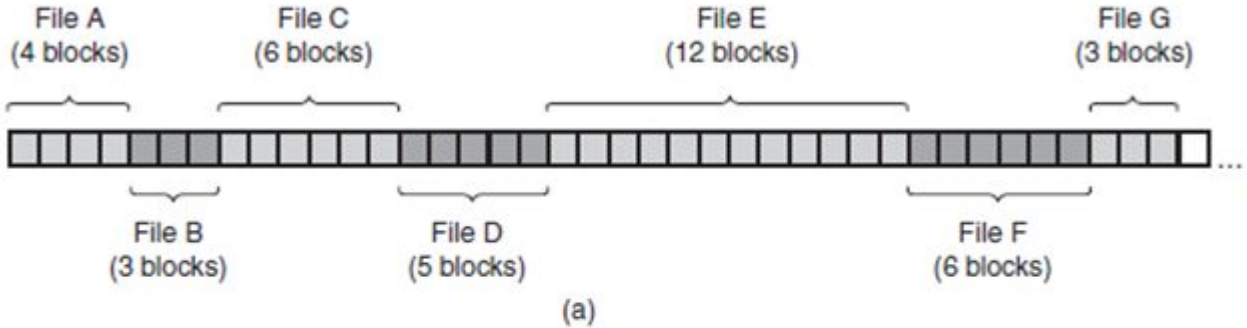
- Filesystems are ultimately the things that determine the structure beyond the Boot-Block
- Often after the Boot-Block comes the *Super Block*
  - This will hold key parameters
  - Size of the rest of the blocks (if in multiples)
  - It’s a vague name, because it’s a vague concept!

Free Space and Files



- The methods of keeping track of free space, directories, and files, is where FS diverge tremendously.
- Let’s first look at Files

Files as Contiguously Allocated Blocks



Contiguous Allocation - good and bad

- Unlike memory, **physical** location tends to affect performance
  - Reading contiguously allocated files (adjacent blocks) is **faster**!
- External Fragmentation:
  - Disastrous compaction time
  - We don’t actually know the size of a file when it’s created!

When do we use Contiguous?

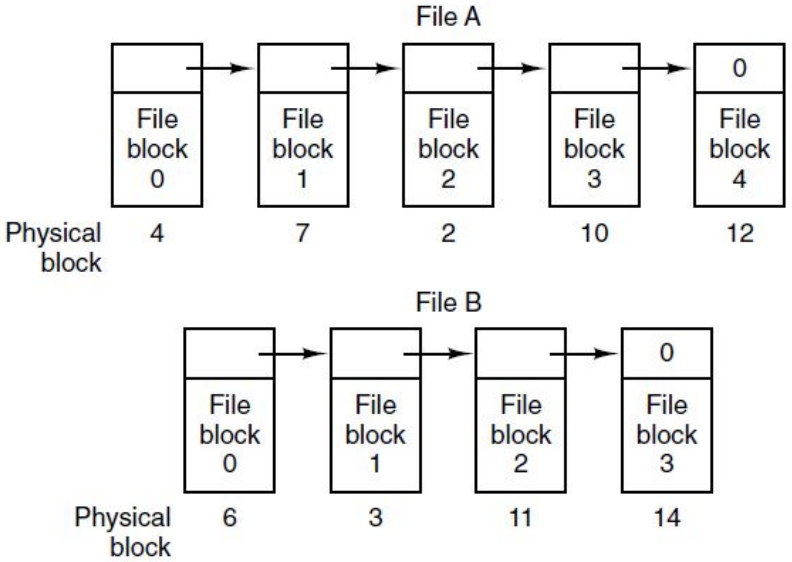
- Unlike memory systems, there are “storage” systems that are used in a “write-once” mode
  - CD-ROMS and DVD’s
  - Backup Tape Drives
- It just so happens, adjacency of blocks is even more important for these - and since external fragmentation is not possible - it's a no brainer!

Contiguous Allocation - Implementation

Each directory must hold the following about each file:

- It’s Name
- The Starting Block
- The Ending Block (or the File Size)
- Likely more… unrelated items (create time...etc.)

Alternatives: Linked List Allocation



- Each block on disk is *part* of a file
- Each block has a header that indicates the block number of the ***next block*** in the file
- The last block's header just has 0

Linked List implementation

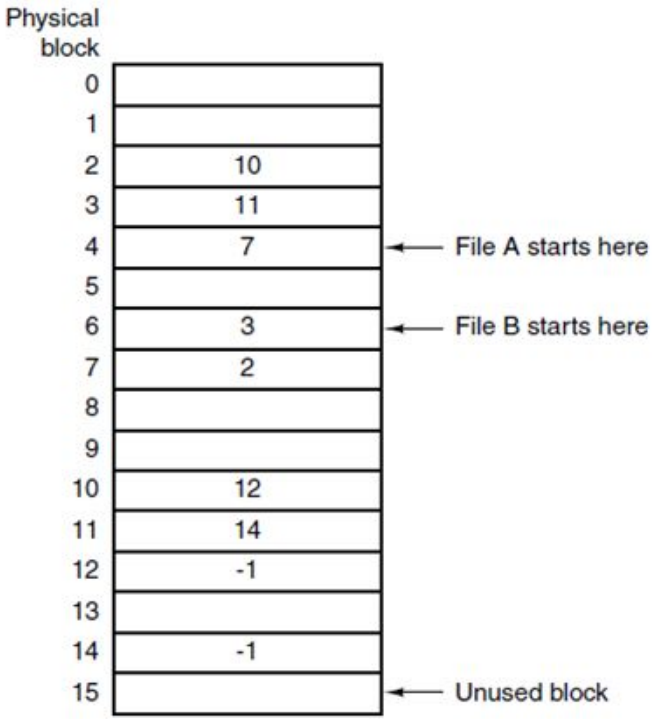
- Directories now simply hold the ***first block***
- However…
  - File size should also be stored - why?
  - Each block now contains “overhead”
  - Can we support random access / seek?

How can we fix random access?

- To support random access, we need **all** the headers - but not their actual data!
- What if we could collect the headers into a single record?
  - File Allocation Table (FAT)

FAT Implementation

- FAT is ***global***. There is an entry for each block on disk.
- A directory holds the first block number for each file within it.
- The FAT must be traversed to get to other blocks within a file…
  - So why is this better?



FAT can be cached!

The FAT is small enough (in theory) to be held in memory… so the FAT can be traversed quickly.

FAT can be cached!

The FAT is small enough (in theory) to be held in memory… so the FAT can be traversed quickly.

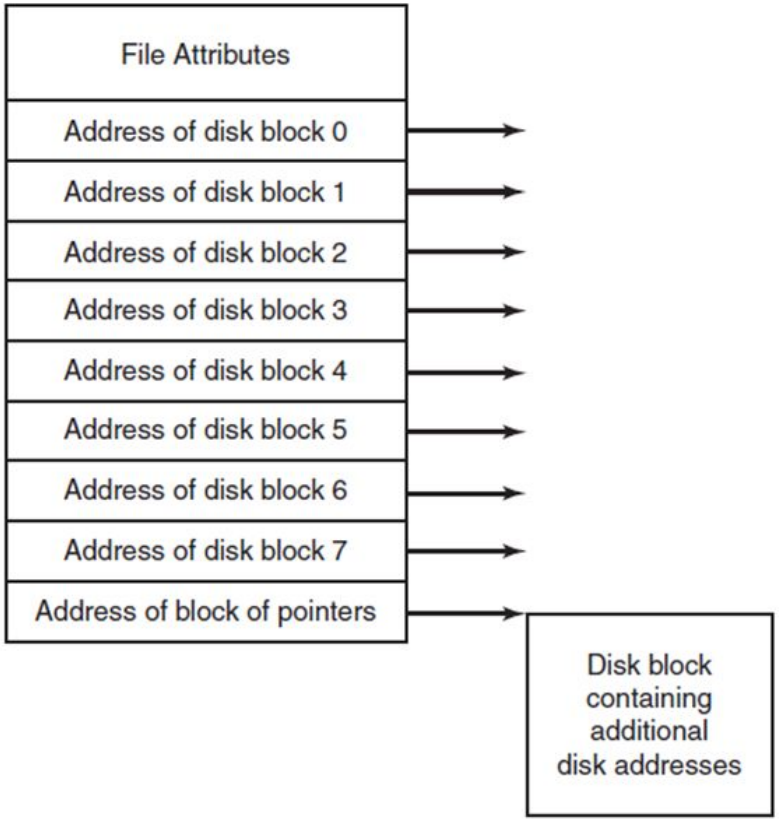
Unfortunately “in theory” is a dangerous phrase in computer science!

Big Disks

- For a 1TB Disk, with 1KB blocks, you have 1 billion entries in the FAT
- Each entry will be 4 bytes (integers)
- This means your FAT is 3GB… meaning 3G of main memory would be used.
  - FAT schemes are no longer practical for modern discs, but are important concepts nevertheless

Distributing the FAT

- A critical problem with FATs is that they are one, giant, data structure.
- i-nodes offer an alternative, where each file has it's own "table"
- Each entry in the table points to the a physical block.
- The model can be extended to support large files by using indirects



i-nodes

Advantages:

- A i-node need only be loaded into memory if the associated file is open
- Unless you are talking about files > TB in size - it’s a win.
- Total memory used is proportional to number of files open, not size of disk.
- Both UNIX and Windows (NTFS) use this system

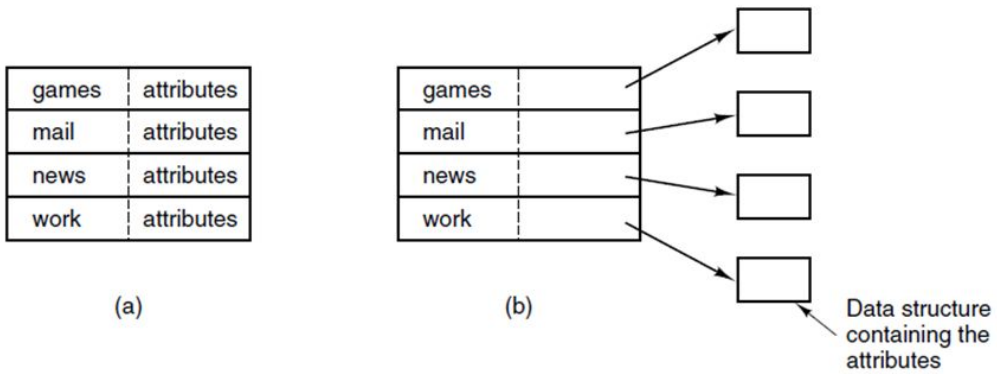
Implementing Directories

- A file path consists of a sequence of directories, ending with (optionally) a file
- Each directory actually represents a ***file*** - stored on disk - which contains information about its files
- We refer to the structure that contains file information is called a **directory entry**.

Directory Entry

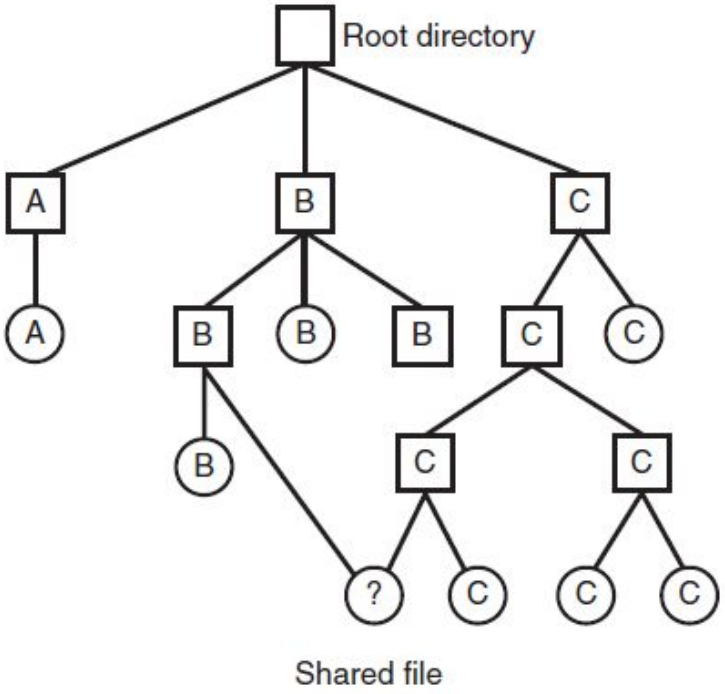
A Directory Entry will contain:

1. File name
2. Block number of first block (linked list/FAT) or block number of i-node (UNIX/Linux/NTFS)
3. File Attributes



Directory entries (and directory files) are often cached by the OS to quicken search and retrieval.

Shared Files



Shared file

Option (b), where attributes are held apart from directory entries, is usually advantageous…

Here's Why!

i-nodes as vehicle for sharing

- i-nodes will be unchanged
  - pre-allocated
  - Assigned when a file is created
- Moving a file updates the i-node data, not the i-node itself
  - i-nodes hold file attributes
  - Directory Entries merely associate name with i-node

Linking

In UNIX, we can link files together via two mechanisms:

- **Hard Link:** Two directory entries associate two different file paths/names with the same i-node
- **Symbolic Link:** A special directory entry is created that simply points to another directory entry.

Log-Structured Filesystems

- File Systems have become a performance bottleneck
  - CPU's keep getting faster (or you have more)
  - Memory is increasing in size - which allows for more disc to be ***cached***.
- Disk Cache's obviously only work for **reads**.
- In the future, more and more **actual** disc accesses will be writes - because reads will be satisfied by cache more often.

Log-Structured File Systems (LFS)

- Writes present a performance problem:
  - Disks are optimized for large chunk operations
  - Write's tend to be small, incremental
- LFS buffers all small write operations
  - … As log entries (updates)
  - … "Change this, Add that.."
  - Periodically, write all log updates to contiguous chunks of disk.

LFS - infinite disks?

- Since LFS is storing incremental changes, the “log” must always grow (deleting a file causes a log entry!)
- LFS must also have a ***cleaner*** mechanism to periodically remove older entries (in a non-destructive way)
- LFS is not used commonly, because it is such a dramatic shift away from standard FS

Journalled File Systems

- LFS presents an interesting idea though - a running history of a file is valuable for one important reason:
  - Computers crash

- Journalled FS keep a log of what they are about to do, before the do it.
  - Once it’s complete, the remove the “log”

Journalling

- To Remove a file in UNIX:
  a. Remove the file from it's directory
  b. Release the i-node so it can be re-used
  c. Return disk blocks to the pool of free blocks
- If we crash in the middle of any of this - we've got big problems!
- Journalling solves this, by keeping the record of all three operations until they've completed

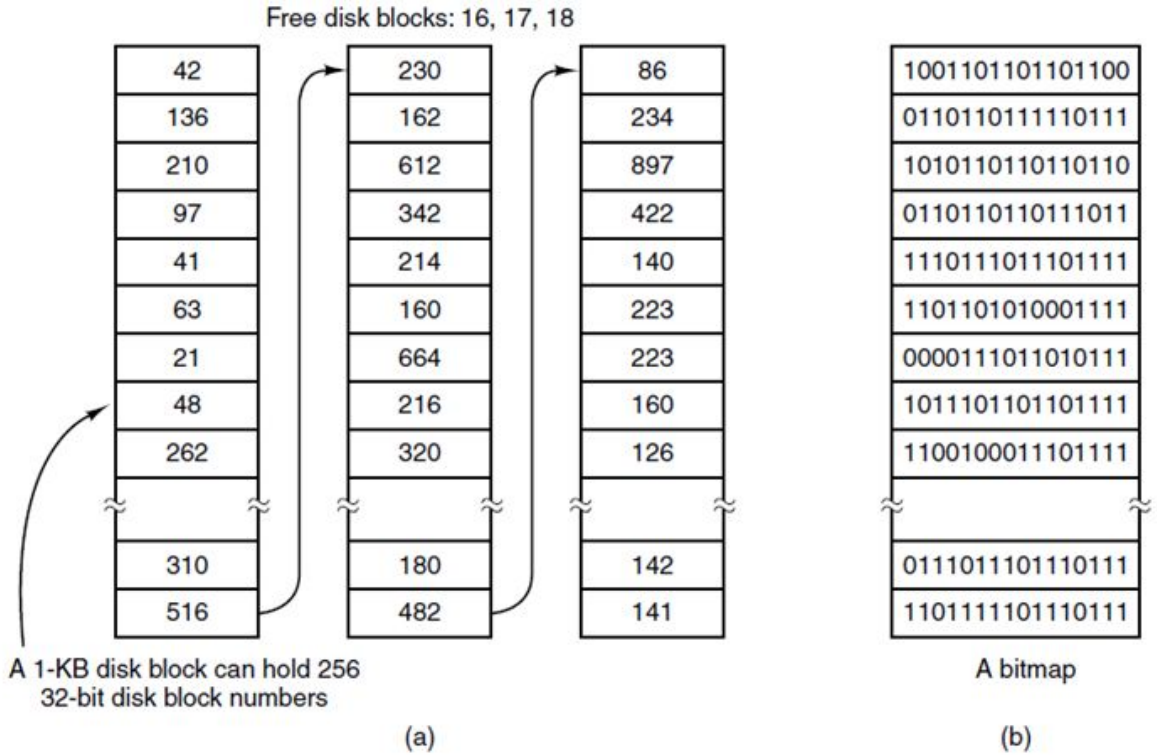
NTFS (1993) using Journaling. Linux with etc3 also uses this strategy

Free blocks?

- We’ve neglected to talk about how the FS can keep track of which blocks are free!
  - It’s actually simple: A “Free” File

- Linked List implementation is usually used, because we don’t need random access! (why?)

Free Blocks


Free disk blocks: 16, 17, 18
42
136
210
97
41
63
21
48
262
310
516
230
162
612
342
214
160
664
216
320
180
482
86
234
897
422
140
223
223
160
126
142
141
A 1-KB disk block can hold 256 32-bit disk block numbers
(a)
1001101101101100
0110110111110111
1010110110110110
0110110110111011
1110111011101111
1101101010001111
0000111011010111
1011101101101111
1100100011101111
0111011101110111
1101111101110111
A bitmap
(b)