CS3503A Lecture # 13 2014-05-13

Administrivia

- Lab 3 posted
- will try to get assignment 3 out by end of week
- no progress grading HW 1
- still accepting HW 2 until Wednesday

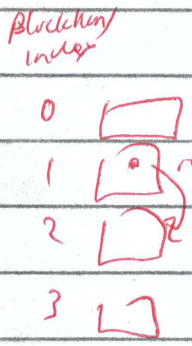
Our Story So Far

- filesystem as data structure
  - file control block aka inode
    - file metadata (creation time/owner/size)
  - multiple filesystem types
  - directory: "special file"
    - unix kernel: just mapping between string (filename) & inode
  - hard link vs soft (symbolic) link
  - superblock
    - where are the inodes (free/allocated blocks)
    - root inode
  - disk allocation
    a) linked list (serial data only)
    b) contiguous block (random access, but fragmentation)

FAT Filesystem

FAT: File Allocation Table
- MS DOS

- table at beginning of filesystem, one entry per block
  → block # is array index
  - contents of table entry:
    pointer to table entry (index) of next block in file



- linked list containing only "next" field
  but pointer ≡ array index ≡ block #
  → block entry is node index

+ no overhead in data blocks
+ faster random access (than linked list)
- multiple disk seeks to read links
  - unless table is cached
  - thumb drive: no seeking

FAT Filesystem (cont.)

FAT16 FAT entry : 16 bits (2 bytes)
- $2^{16} = 64k$

block size : 32k
- 128 kb for table $\Rightarrow$ 4 blocks

total disk size :

$$64k \times 32k = 2^6 \times 2^{10} \times 2^5 \times 2^{10}$$
$$= 2^{31} \text{ bytes}$$
$$\Rightarrow 2\,GB$$

FAT 32
- 28 bits of 32 entry size — 256 M entries $2^8 \times 2^{20}$
- 32 kb block size
- limit 2T (overload)

~~$2^{40} \simeq (10^3)^6 \simeq 10^{18}$~~

$2^{28} \times 2^{15} = 2^{43}$

Linux Filesystem

ext2

Block size:	1k	2k	4k	8k
max file size	16G	256G	2TB	2PB
max filesystem	4T	8	16	32

- many userland programs cannot handle files larger than 2 GB

$2^{60} \sim 2^{18}$

ext4

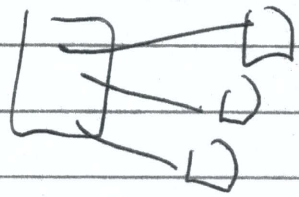
up to 1 exbibyte filesystem

up to 16 TiB file — 4k block size

$\sim 16 \times 10^{12} = 16$ trillion

Indexed File Allocation

index block: pointers to disk blocks



- index block(s): per file
  (cache FAT)

- index block size:
  - too small: cannot support large files
  - too large: waste disk space

- multi-level indexed allocation
  - e.g. unix inode

Unix Inode Indexing

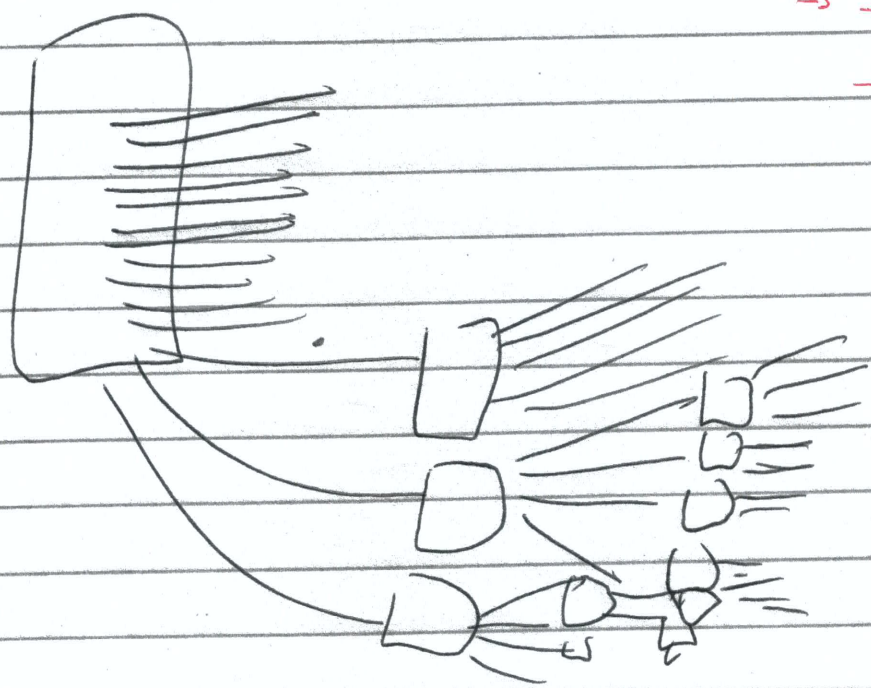
Mode (permissions)
Owner/Group
timestamps (3)
size, block count

12 direct block pointers
1 pointer to indirect block (single indirection)
1 pointer to double indirection
1 pointer to triple indirection

⇒ fast for small files
support for large files

4k bytes per block

128 pointers per block
→ 32 bytes per block address
→ 128 bits



Journaling Filesystems

- journal: transaction log
  - if filesystem crashes, recover incomplete operations from journal

Free Space Management

- bit vector
  - ⊕ easy to find contiguous space
  - ⊖ expensive use of memory
- linked list
  - ⊕ space efficient
    - use free space
  - ⊖ hard to find contiguous space

⇒ essentially, a memory management problem

RAID → redundant array of inexpensive disks

Raid 0
- even split "Stripes"
- no parity info → no fault tolerance
  - failure of one disc loses all data
- benefit: speed
- large "logical" volume

Raid 1
- mirrors
- no parity info
- duplicate writes
  - slow
- Faster reads

Raid 2/3
- not used much in practice

Hamming code
Hamming code parity

Raid 4
- block-level striping
- dedicated parity disk
  writing bottleneck

(Reed Solomon?)

Raid 5

- block-level striping
- distributed parity
- tolerates failure of single drive
  - lost data reconstructed from parity
- faster read performance (than raid 4)

Raid 6

- Raid 5 + additional parity block
- tolerates 2 concurrent drive failures
- no performance penalty for read
- slower writes (additional parity block)

Hamming Code

- error correction / detection
  - single-bit correction
  - double-bit detection

- parity bit: single-bit detection

- Hamming (7,4) code: 4 data bits & 3 parity bits
  → correct single-bit error
  add overall parity bits to get double-bit detection

Parity : even / odd
→ detect 1-bit flip

Hamming Code : (7, 4)
7 bits to encode 4
→ 3 "check bits" - analogous to Parity
- Correct any single bit flip
Detect any 2 bit changes

} Parity on overlapping subsets

- number the bits from 1
- place parity bits when bit # is a power of 2
- each parity bit is responsible for bits that have 1-bit in corresponding position (of bit#)

111	110	101	100	11	10	1
a	b	c	P_3	d	P_2	P_1

a :	P_1, P_2, P_3	P_1 :	a, c, d
b :	P_2 P_3	P_2 :	a, b, d
c	P_1 P_3	P_3 :	a, b, c
d	P_1 P_2		

Single-bit error in data bits will cause unique parity error from which we can determine the error bit

Single-bit parity error will be detected because other parity bits will be correct