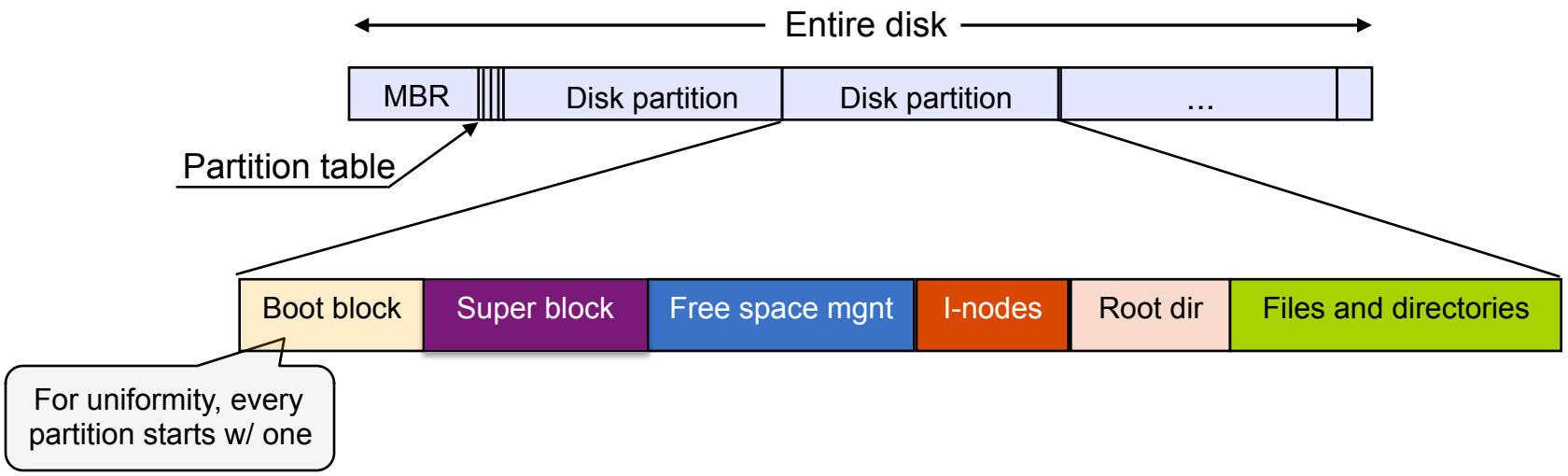
File Systems Management and Examples

To do ...

- Some early file systems
- Efficiency, performance, recovery
- Next: Distributed systems

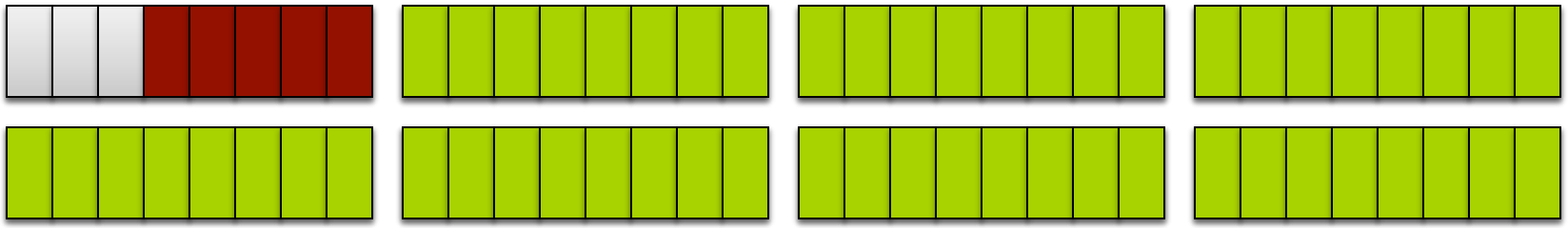
File system layout

- Disk divided into partitions – 1 FS per partition
- Sector 0 of disk – MBR (Master Boot Record)
- Followed by Partition Table (one active)
  - (start, end) per partition; one of them active
- Booting
  - BIOS → MBR → Active partition's boot block → OS
- What else in a partition?



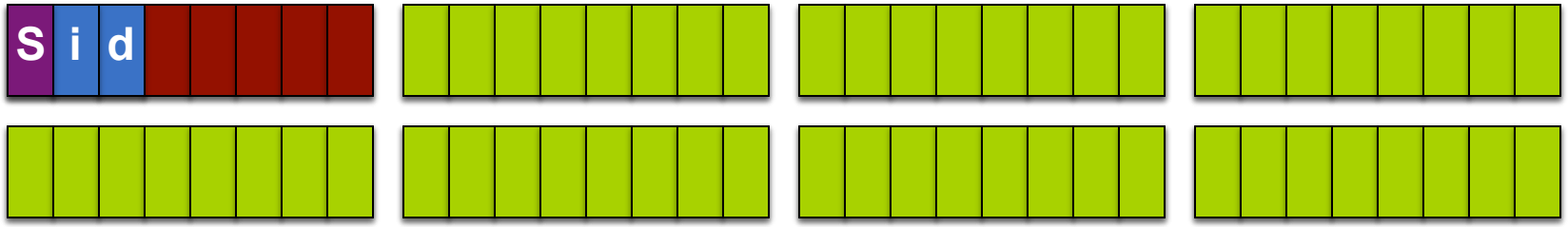
A basic file system

- A small disk with only 256KB
  - 64 4KB blocks
  - To store user data, 56 of the blocks
  - To keep metadata about each file store, 5 blocks for i-nodes
    - Multiple i-nodes per block (they are not too big)



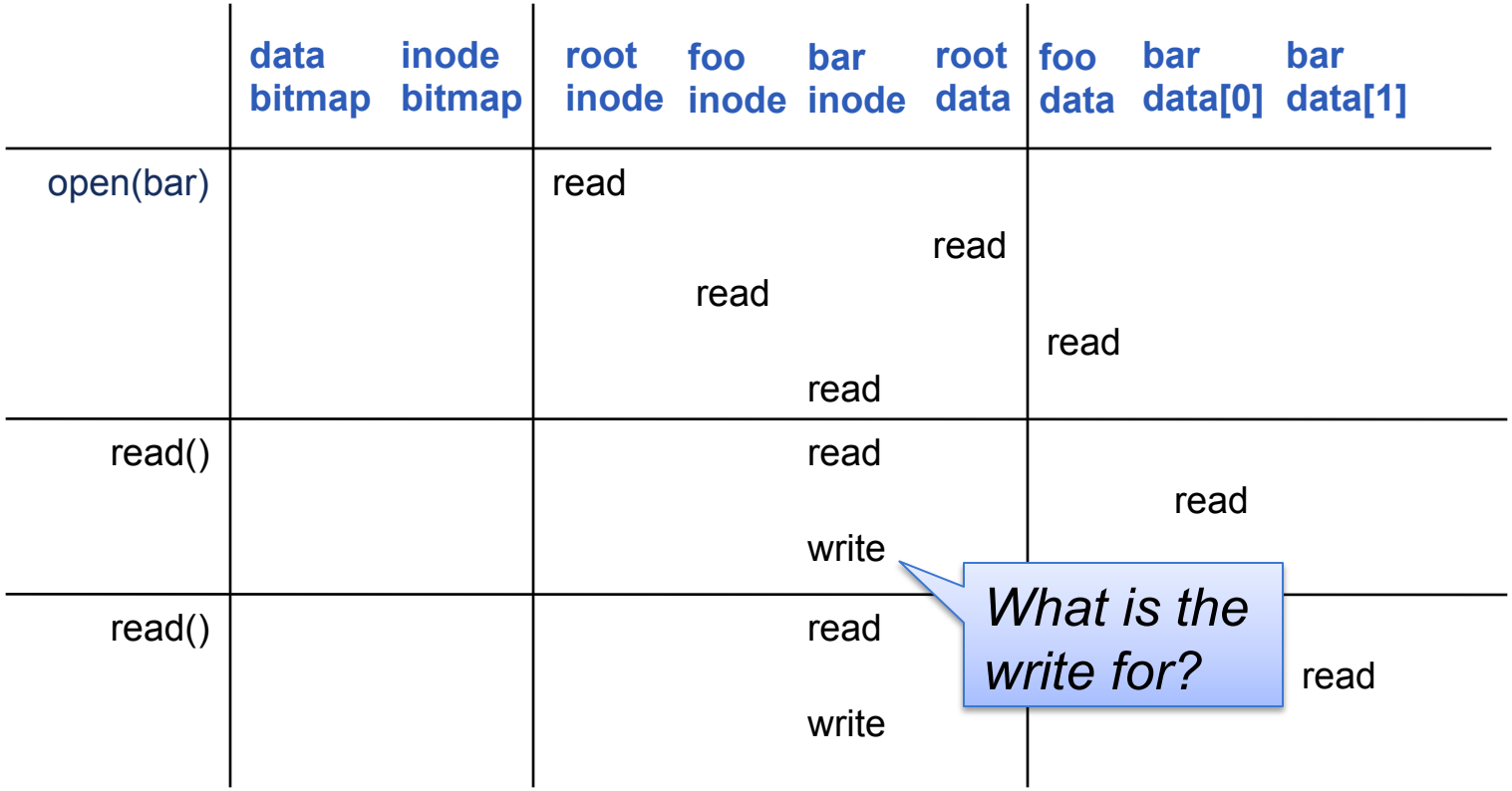
A basic file system

- A small disk with only 256KB
  - Some way to track free and allocated blocks, and i-nodes - bit map: an i-bmap and a d-bmap
  - And to keep information about the file system, another block – the superblock
    - Includes magic number, number and size of blocks, …
    - Read into memory at boot or when FS first touch



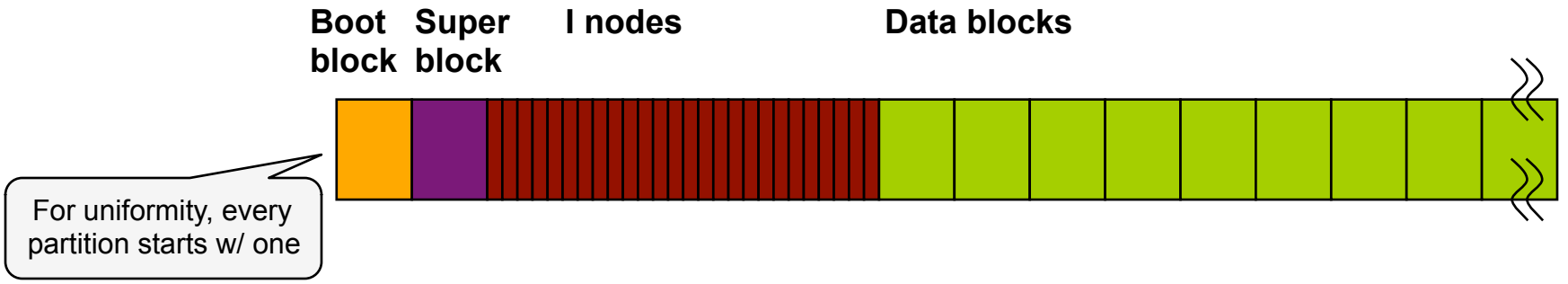
Reading a file from disk

- Open and read a file `/foo/bar`
  - To get started, the root inode must be well known
  - Usually you find the inode of a file or directory in its parent directory



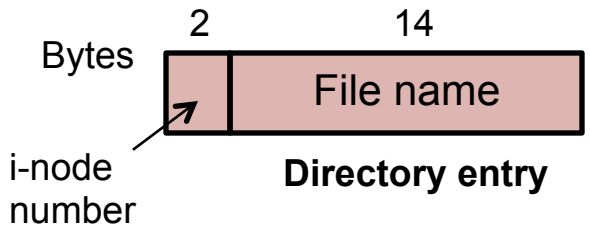
The UNIX V7 file system

- Unix V7 on a PDP-11
- Disk layout in classical UNIX systems
- Superblock
  - Information about the entire file system (size of the volume, number of inodes, …)
  - And pointer to head of a free list of blocks
- The i-node region has all the i-nodes of the disk
- After that – data blocks in no particular order



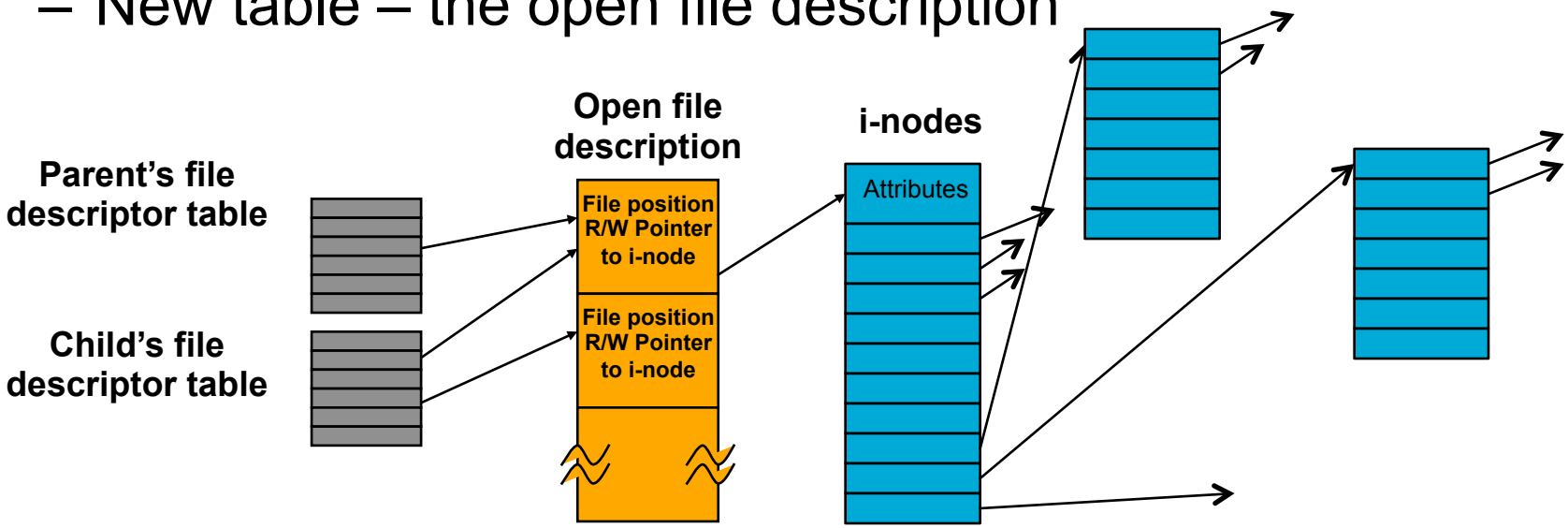
The UNIX V7 file system

- Each i-node – 64 bytes long
- I-node's attributes
  - File size, three times (creation, last access, last modif.), owner, group, protection info, # of dir entries pointing to it
- Tree structured as a DAG
- File names ≤ 14 chars (anything but "/", NUL)
- A directory – Unsorted set of 16-bytes entries



The UNIX V7 file system – Accessing a file

- Starting from file descriptor, get the i-node
  - A file descriptor table indexed by FD
  - To get to the i-node, pointer to i-node in FD table?
  - No; associated with each FD there's a current pointer (CP), *where would you put it?*
    - i-node – But multiple processes each with their own CP
    - FD table? No, parent and children cannot share it (">")
  - New table – the open file description



The Fast(er) File System

- The original Unix FS – Good and simple
- The only problem – Performance
  - Things could get as bad as to have the FS delivering 2% of overall disk bandwidth
- Major problem
  - Treat the FS as RAM with data spread all over – i-nodes, file blocks – as the disk space got fragmented

Files' blocks

After a few file deletes

Red file is all over now

- Also – Original block size was too small – 512B
  - Low internal fragmentation but poor disk performance

FFS – Mind the disk

- For a faster FS, design it to be “disk aware”
- And a new approach to file systems research
  - Keep the same interface
  - Change the implementation
- Organization, policies – reduce disk arm motion
  - Disk divided into cylinder groups
    - Linux ext2 and ext3 called them block group)
    - A copy of the superblock per group
    - Each with its own blocks & i-nodes
    - To allocate according to cylinder groups you need enough free space → save 10% of the disk just for that!

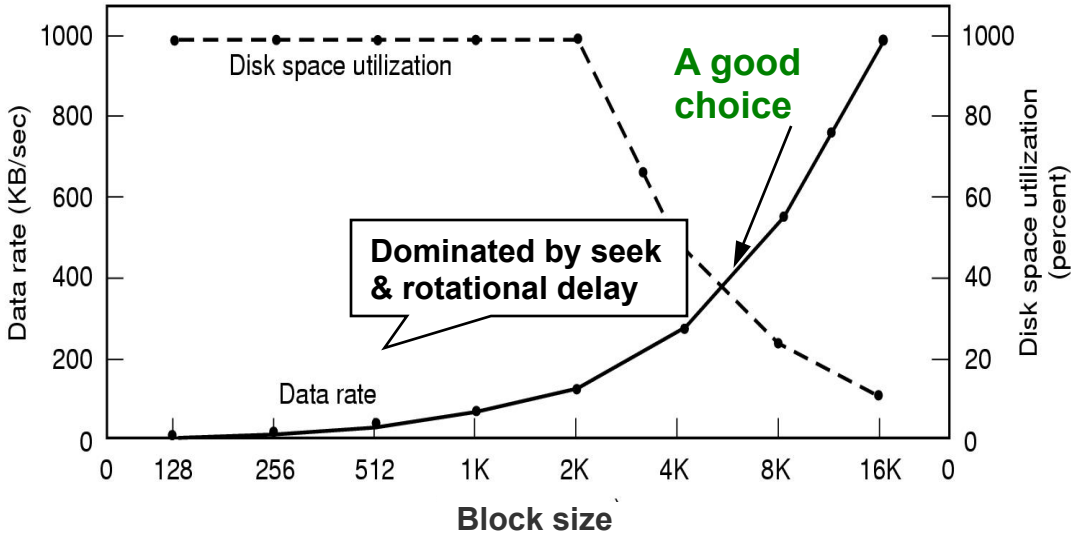
FFS – Mind the disk

- Keep related stuff together
  - Put blocks likely to be accessed in sequence close
- For directories
  - Find cylinder group with a low number of allocated directories, put directory data and i-nodes there
- For files
  - Allocated data blocks and i-nodes in same group; place all files in a directory in same cylinder group of the directory
- An exception for large files
  - Past first twelve direct blocks, all indirect blocks and those they point to go in different groups

FFS – Help with the right block sizes

- What's the right size of the block?
  - Candidates: Sector, track, cylinder, page
  - Decide base on median file size (instead of mean)
  - Performance and space utilization are in conflict

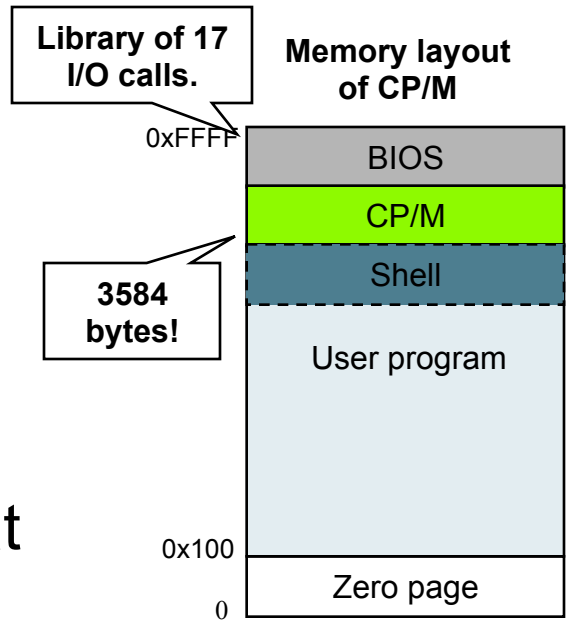
Assume all files 2KB

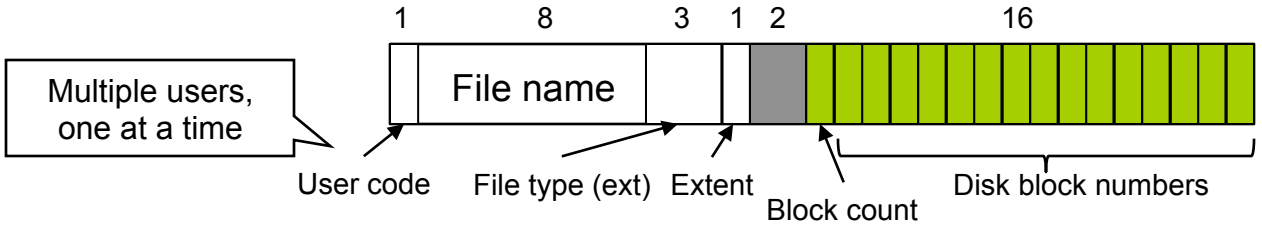


- FFS handling of small files
  - Sub-blocks (512B) and buffering

Older PC examples – CP/M file system

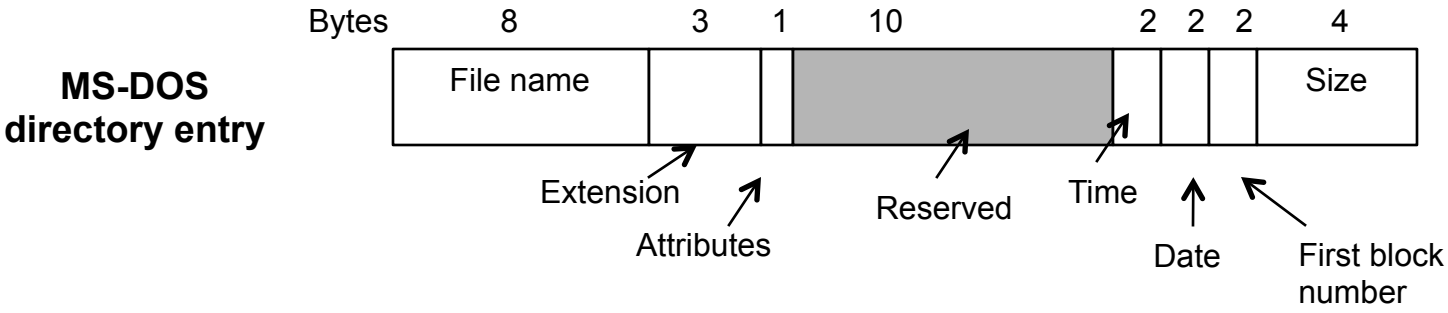
- Control Program for Microcomputers
- Run on Intel 8080 and Zilog Z80
  - 64KB main memory
  - 720KB floppy as secondary storage
- Separation between BIOS and CP/M for portability
- Multiple users (but one at a time)
- The CP/M (one) directory entry format
  - Each block – 1KB (but sectors are 128B)
  - Beyond 16KB – Extent
  - (soft-state) Bitmap for free space





Older PC examples – MS-DOS file system

- Based on CP/M; used by the iPod
- Biggest improvement: hierarchical file systems (v2.0)
  - Directories stored as files – no bound on hierarchy
  - No links – so basic tree
- Directory entries are fixed-size, 32 bytes
- Names shorter than 8+3 characters are left justified and padded



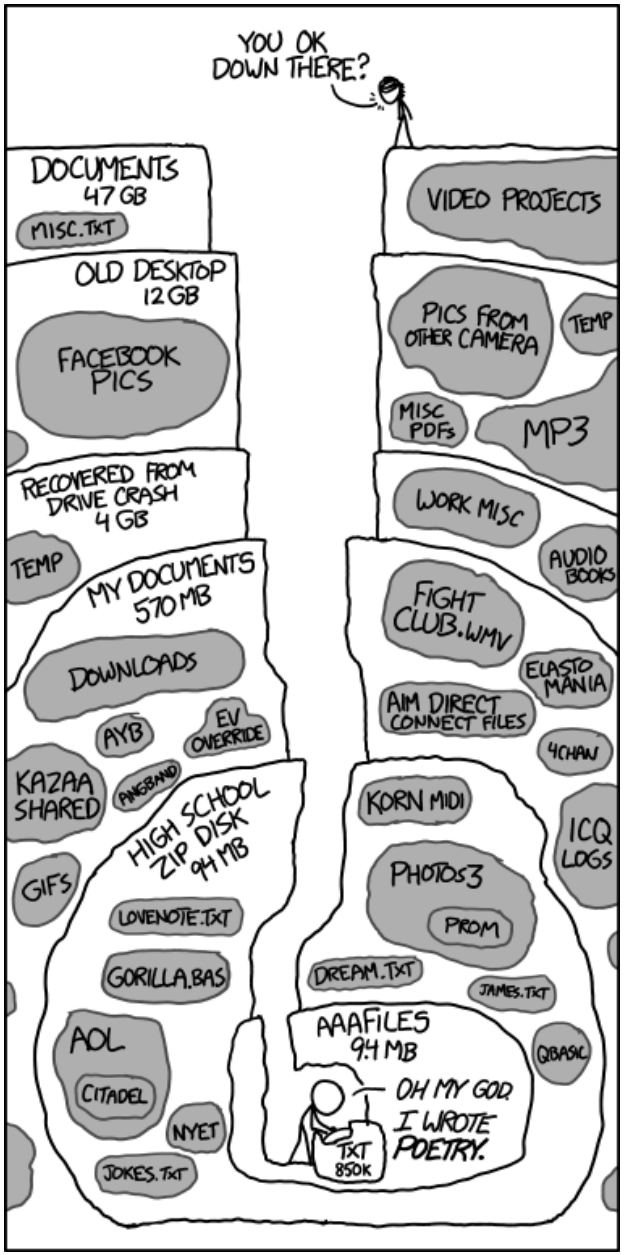
The MS-DOS file system

- Attributes include:
  - Read-only – to avoid accidental damage
  - Hidden – to prevent it from appearing in a listing
  - Archived – used by, for instance, a backup system
  - System – Hidden and cannot be deleted using ‘del’
- Time – 5b for secs, 6b for min, 5b for hours
  - Accurate only to +/-2 sec (2B – 65,536 sec of 86,400 sec/day)
- Date – 7b for year (i.e. 128 years) starting at 1980 (5b for day, 4b for month)
- Size is a 32bit number (so, theoretical up to 4GB files)

The MS-DOS file system

- Another difference with CP/M – FAT
  - First FAT-12: 12bit disk addresses & 512B blocks
  - Maximum partition 2^{12}x 512 ~ 2MB
  - FAT with (2^{12}) 4096 entries of 2 bytes each – 8KB
- Disk block sizes can be set to multiple of 512B
- Later versions
  - FAT-16, FAT-32 (only the low-order 28-bits are used)
- FAT-16
  - 128KB of memory for the FAT table
  - Largest partition – 2GB ~ with block size 32KB
  - Largest disk - 8GB (limit of 4 partitions per disk)
- *How do you keep track of free blocks?*

And now a short break …

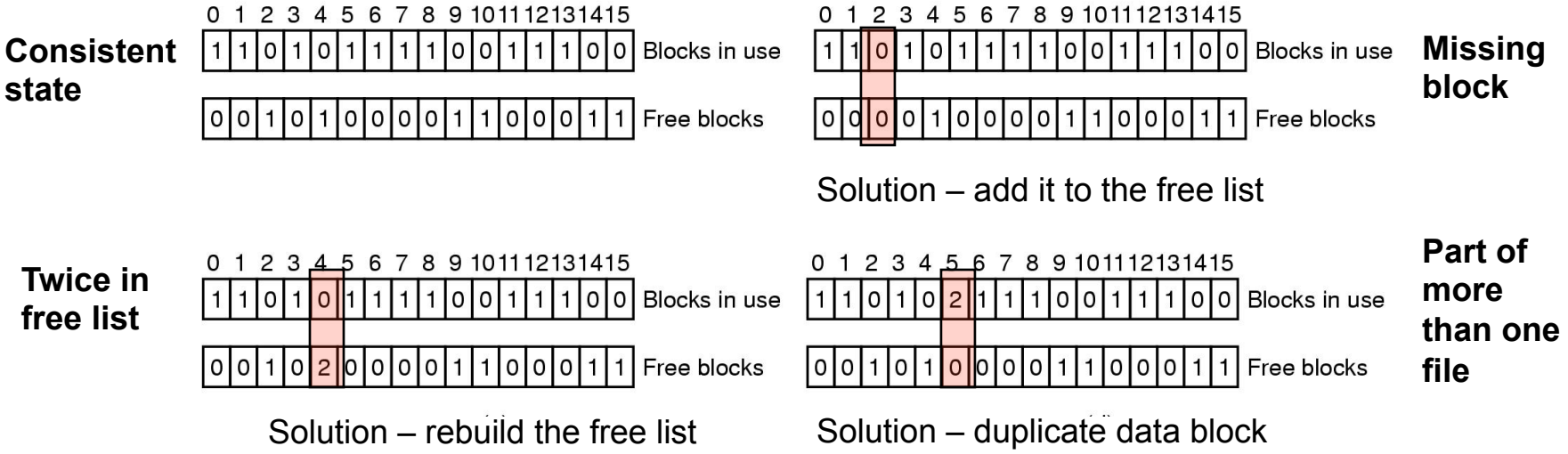


File system reliability – crash consistency

- The FS manages a set of persistent data structures to implement its abstractions
  - What if system crashes or power fails while updating them? Crash consistency problems
- Consider a simple file removal (in UNIX)
  1. Remove the file from its directory
  2. Release the i-node to the pool of free i-nodes
  3. Return all disk blocks to the pool of free disk blocks
- If only (1) succeed before the system crashes
  - i-node and free blocks will be lost in limbo
- If (1) and (2) succeed – Lost free blocks
- Alternative orderings don't help

fsck – The file system checker

- Basic early approach – let inconsistency happen and fix them later, when rebooting (lazy approach)
- fsck basic ideas
  - Run before the file system is mounted
  - Check if the superblock looks reasonable, e.g. size of the file system and blocks allocated
  - Free and used blocks check; Build two tables – a counter per block and one pass; every block should be in exactly one list

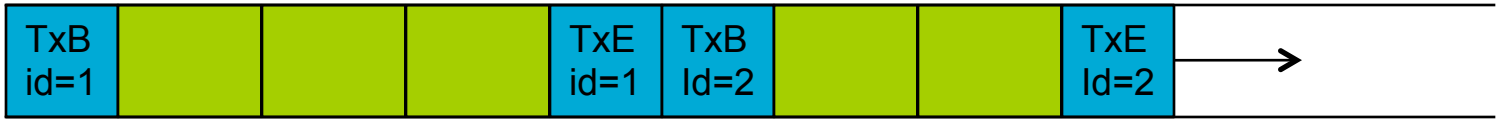


fsck – The file system checker

- Similar check for directories – link counters kept in i-nodes
- Overall, a lot of work
  - Requires a thorough understanding of the file system
  - Very very slow, getting worst with growing disks!
  - A bit of a brute-force approach – Maybe a couple of writes have gone bad, but we scan the *whole* disk
- A more focused approach …

Journaling file systems

- Basic idea from databases
  - Update metadata (and possibly all data), transactionally – *all or nothing*
- How? Write-ahead logging, journaling in OS
  - Log your actions ahead
    - For each transaction – Start and end of transaction, actual blocks for physical logging



  - Once on disk, do what you planned
  - If actions complete successfully, remove the log
  - If not, rerun from the log

Journaling file systems

- How? Write-ahead logging or journaling in OS
  - …
  - Of course, logged operations must be (made) idempotent
    - Valid transaction if has a ‘end of transaction’
    - Disk guarantees that small writes (512B) are atomic
  - Log is of a fixed size, so free previous entries
- If a crash occurs, you may lose a bit, but the disk will be in a consistent state
- There are several options in use
  - Cedar (first one ~ ’87), Ext3, Ext4, ReiserFS, NTFS

Log-structured file systems

- CPUs getting faster, memories and disks larger
  - But disk seek time lags behind
  - Since disk caches are also getting larger → increasing number of read requests come from cache
  - *Thus, most disk accesses are writes*
- To make matter worse
  - Writes are normally done in very small chunks
    - To create a new file, writes for: directory i-node, directory block, file i-node, and file's blocks
    - RAID-4 and RAID-5 have the small write problem too
  - Low disk efficiency, most of the time gone on seek and rotational delay

LFS – Disk as a log

- LFS strategy - structure entire disk as a log to achieve the full bandwidth of the disk*
- All writes initially buffered in memory
  - In a *segment*
- Periodically write segment to end of disk log
  - Each segment may contain, all mixed together, i-nodes, directory blocks, and data blocks
  - Each segment has a summary at the start
  - If the segment can be made to be large enough, we can use the full bandwidth of the disk

*M. Rosenblum & J. Ousterhout, “The design and implementation of a log-structured file system”, Proc of SOSP, 1991

LFS – Disk as a log

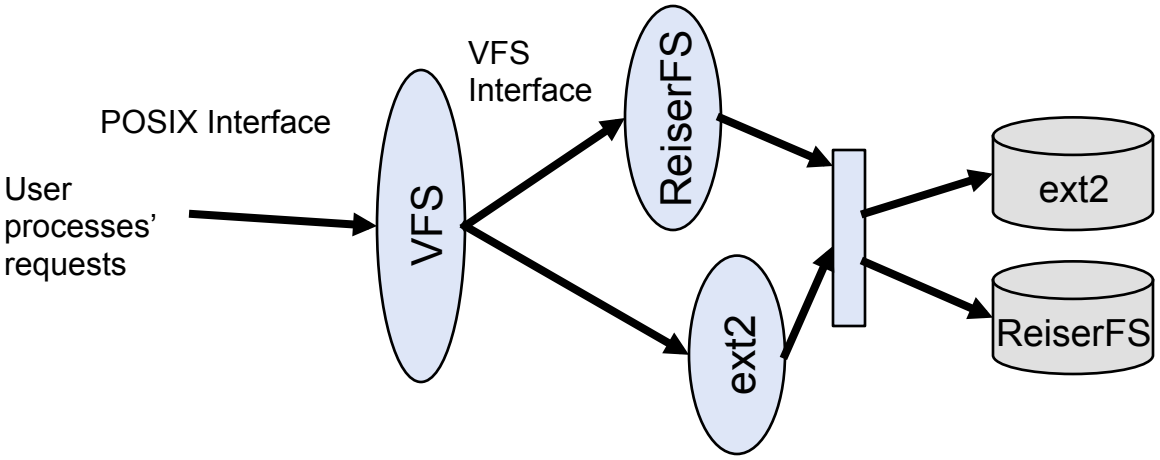
- When file opened, locate i-node, find blocks
  - i-nodes are not at a fixed position but scattered
  - Keep i-node map in disk, index by i-node, cache it
    - *Where do you write the i-node map?*
- To deal with finite disks: cleaner thread
  - Files overwritten, deleted … compact segments from the front
  - Read summary, check with current i-node map if i-nodes are in use, create a new segment, mark the old one free

Virtual file systems

- Many machines use multiple FS at once
  - Windows – NTFS, legacy FAT, CD-ROM, …
  - Unix – ext2. ext3, NFS
  - Rather than exposing this to the user (Windows, using drive letters), UNIX integrates them

Virtual file systems

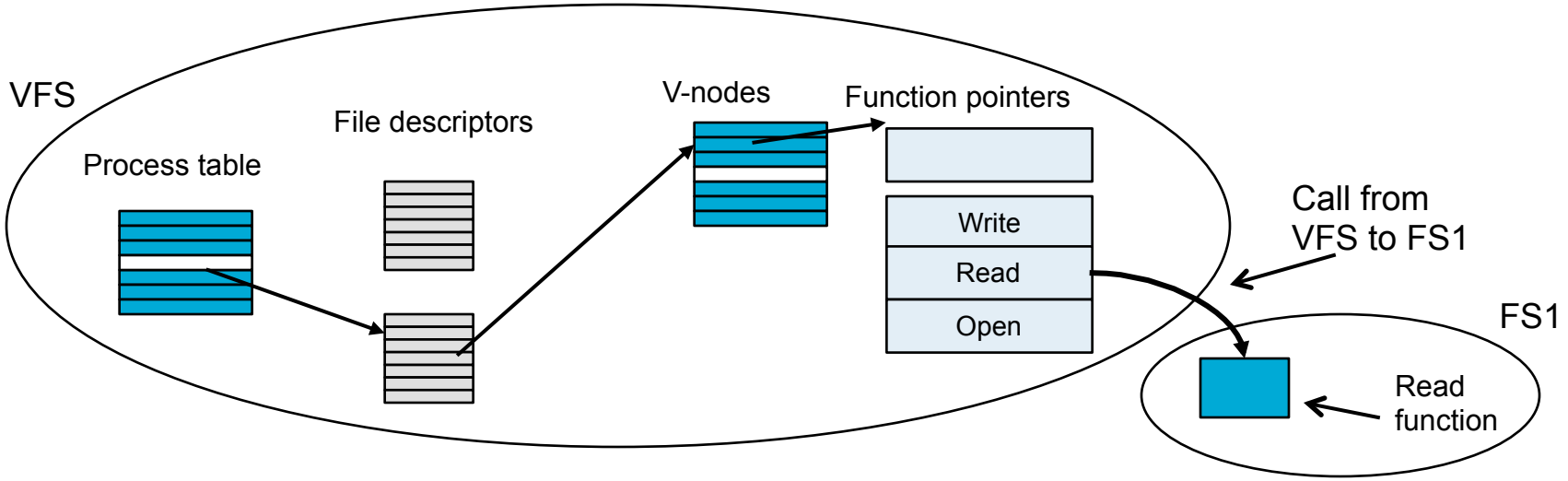
- Virtual file system (VFS)*
  - Key idea: abstract what is common to all FSs
  - Exposes the POSIX interface to user processes
  - FSs implement calls that the VFS interface will invoke



*S. Kleinman, "Vnodes: An architecture for multiple file ystem types in Sun UNIX", Proc. of USENIX Summer 1986

Virtual file systems

- Exposes the POSIX interface to user processes
  - Maintains key objects including superblock (describing the FS), v-node (describing the file) and directory
- FSs implement calls that the VFS interface will invoke
- When FS is registered, at boot time or when mounted
  - Provide list of addresses to the functions VFS requires
  - When a file is created, a v-node is created (in RAM) with data from the i-node and pointers to the table of functions



Next Time

- ~~Multiple processors, distributed systems and …~~
- research in operating systems

File system reliability – backups

- Need for backups
  - Bad things happen & while hardware is cheap, data is not
- Backup - needs to be done efficiently & conveniently
  - Not all needs to be included – /bin?
  - Not need to backup what has not changed – incremental
    - Shorter backup time, longer recovery time
  - Still, large amounts of data – compress?
  - Backing up active file systems
  - Security

File system reliability

Strategies for backup:

- Physical dump – from block 0, one at a time
  - Simple and fast
  - You cannot skip directories, make incremental backups, restore individual files
- Logical dumps
  - Keep a bitmap indexed by i-node number
  - Bits are set for
    - Modified files
    - Every directories
  - Unmarked directories w/o modified files in or under them
  - Dump directories and files marked
  - Free list is not dump, but reconstructed