File Systems

Today

- Files and file systems
- File system structure
- File & directory implementation
- Efficiency, performance, recovery
- Examples

Next

- Distributed systems

Files and file systems

Most computer applications need to:

- Store large amounts of data (larger than their address space)
- that must survive process termination and
- can be access concurrently by multiple processes

→ Usual answer: Files – form user's perspective, the smallest allotment of logical secondary storage

File system – part of the OS dealing with files

- Supports the file abstraction of storage
- Naming – how do users select files?
- Protection – users are not all equal
- Reliability – information must be safe for long periods of time
- Storage mgmt. – efficient use of storage and fast access to files

File operations

- File is an ADT (Abstract Data Type) – what operations?
  - Create, delete, write, read
  - Reposition within file – file seek
  - Truncate
- Most operation involve searching the directory for file
  - Open (F_i) - search directory for entry F_i, move content to memory (open-file table)
  - Close (F_i) - move F_i content in memory to directory structure on disk
- Open/Close in multiuser systems
  - Per-process and system-wide tables
    - Entry in the per-process table points to system-wide table
  - Open counts
- File locks – restricting access to a file
  - Shared (read) and exclusive (write) locks
  - Mandatory (OS enforced) and advisory (cooperative model) locks
  - Lock files

File attributes

- Names – different for each OS
  - Upper and/or lower case
- Support for file extensions or just convention (Unix)
- Some typical file extensions

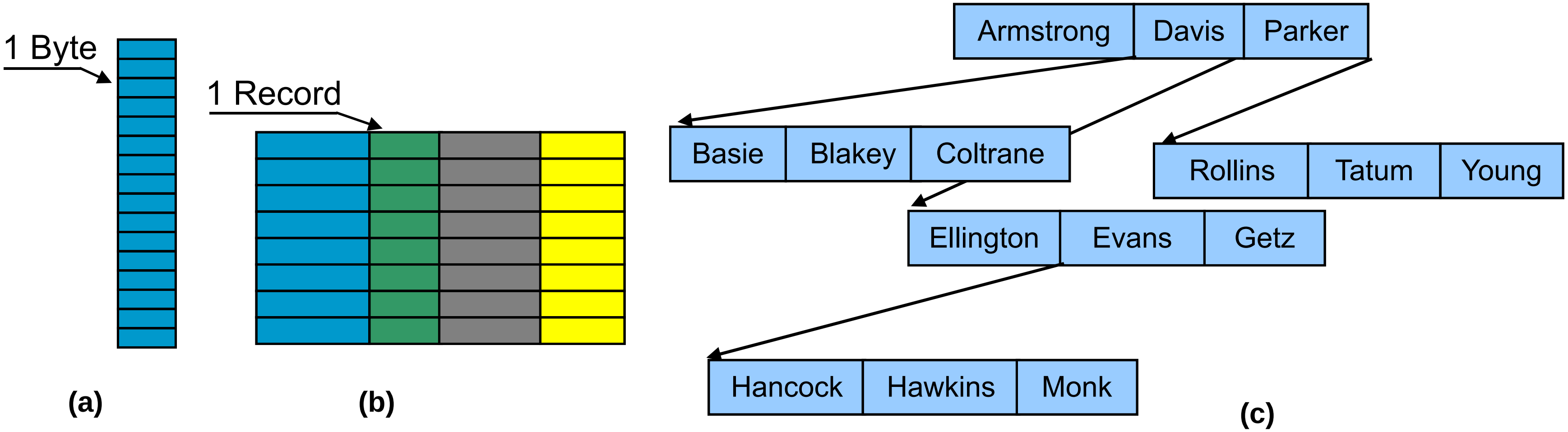
file.gif	Graphical Interchange Format Image
file.mpg	Movie encoded with MPEG standard
file.o	Object file
file.txt	General text file

- A few other useful attributes

Protection	Who can access the file & in what way
Creator	ID of creator
System flag	0 for normal files; 1 for system ones
Creation time	Date & time of creation
Time of last access	Date & time of last access
Current size	In bytes

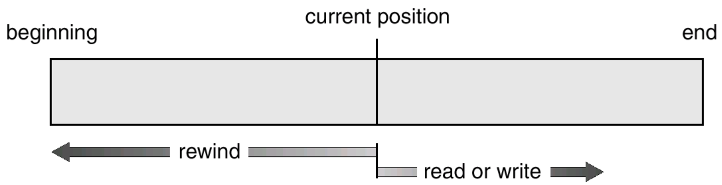
File types and structures

- Different OSs support different file types
  - Regular, binary, directories, … (example of TOPS-20)
  - Extensions as hints & the use of magic numbers
  - Pros and cons of strongly typed files
- Several file structures, three common ways
  - Byte sequence - Unix & Windows; user imposes meaning (a)
  - Record sequence – think about 80-column punch cards (b)
  - Tree – records have keys, tree is sorted by it (d)

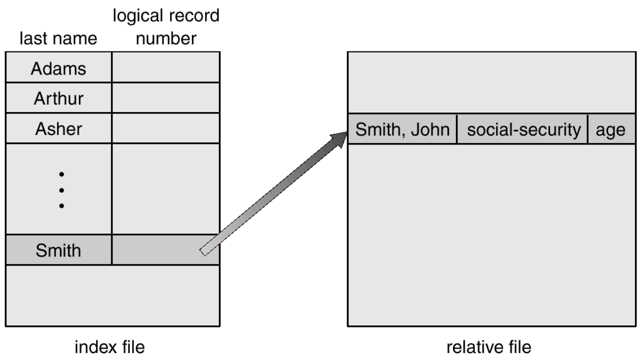


File access methods

- Sequential Access – tape model
  - Simplest and most common
  - read next/write next
  - reset/seek



- Random/direct access – disk model
  - read n/write n, position to n and read next/write next
  - rewrite n (n = relative block number)
  - Retain sequential access – read/write + update last position
- Other access methods
  - On top of direct access
  - Normally using indexing
  - Multi-level indexing for big files
    - E.g. IBM ISAM (Indexed Sequential Access Method)

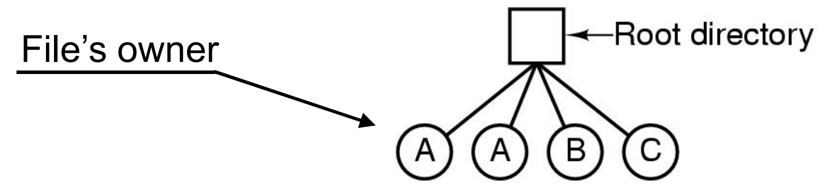


Directory structure

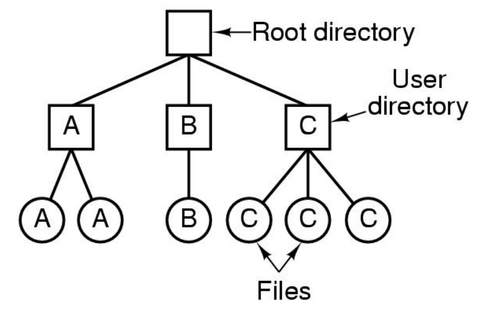
- To manage volume of info.: partitions & directories
- Directory: set of nodes with information about all files
  - Name, type, address, current & max. length, date last accessed
- Operations on directories
  - Open/close directories, create/delete/rename files from a directory, readdir, link/unlink, traverse the file system
- Directory organizations - goals
  - Efficiency – locating a file quickly.
  - Naming – convenient to users.
  - Grouping – logical grouping of files by properties (e.g. all Java progs., all games, …)

Single and two-level directory systems

- A single level directory system
  - Early PCs, early supercomputers (CDC 6600), embedded systems?
  - Pros and cons
    - Fast file searches
    - Name clashing
  - Contains 4 files owned by 3 != people

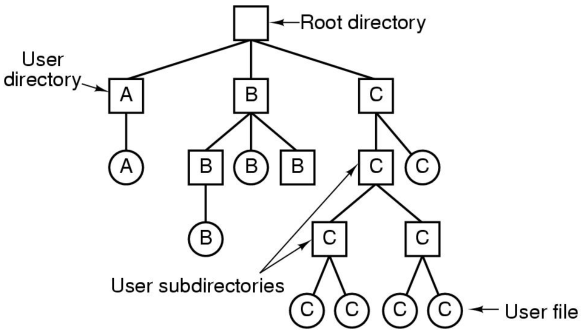


- Two-level directory system
  - Avoid name conflicts bet/ users
  - You may need a system’s directory
  - Problems if you have too many files

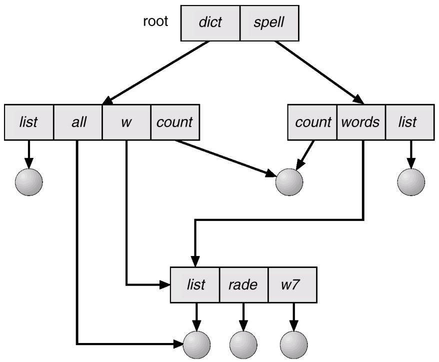


Hierarchical & general directory systems

- Hierarchical
  - Avoid name clashing for users (MULTICS)
  - Powerful structuring tool for organization (decentralization)



- Acyclic graphs – sharing
  - Two different names (aliasing)
  - If *dict* deletes *list* → dangling pointer
    - Backpointers & counter
  - Unix links – pointers to files
    - Soft & hard links – (in)direct pointer



- Path names
  - Absolute & relative path names
  - "." & ".."

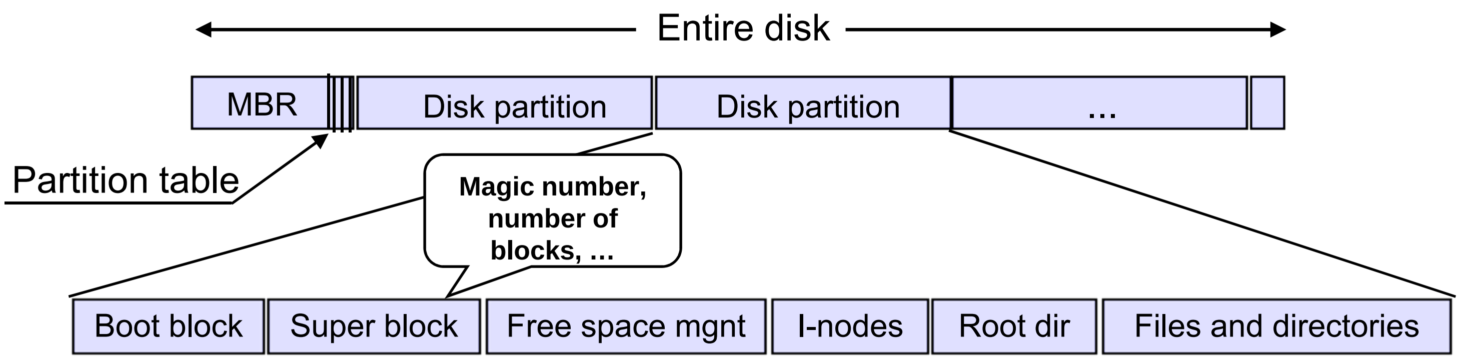
Protection

- File owner/creator should be able to control
  - what can be done & by whom
- Types of access
  - Read, Write, Execute, Append, Delete, List, …
- A general & common approach – access control list (ACL)
  - Per resources – user names & types of access allowed
  - Long!
- Unix: short version access lists & groups
  - Access modes : read, write, execute
  - Classes of users: owner, group, public
  - 3 bits per for each access mode
  - Mask provides a default (mine ‘022’ - octal)
  - File created with 777 and mask 022 $\rightarrow$ 755

Rights	**Code**
rwx	7 (111)
rw-	6 (110)
r-x	5 (101)
r--	4 (100)
-wx	3 (011)
-w-	2 (010)
--x	1 (001)
---	0 (000)

File system layout

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

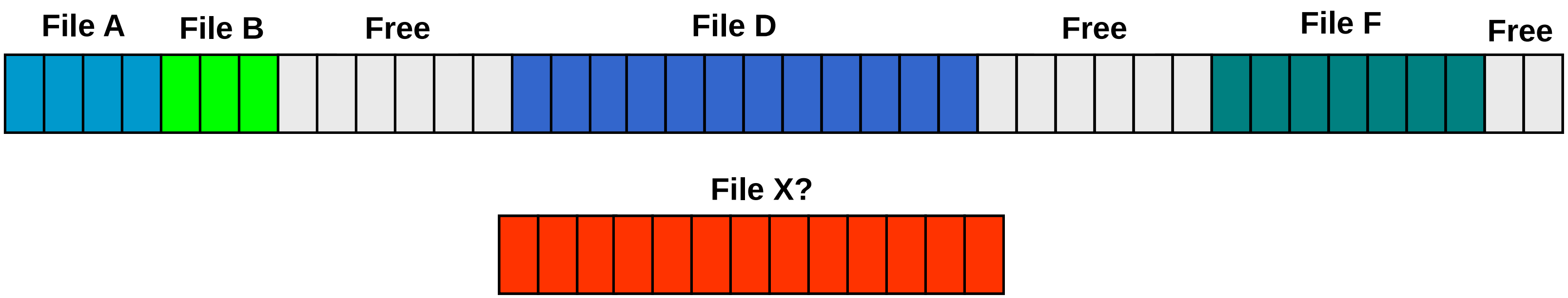


Implementing files

Keeping track of what blocks go with which file

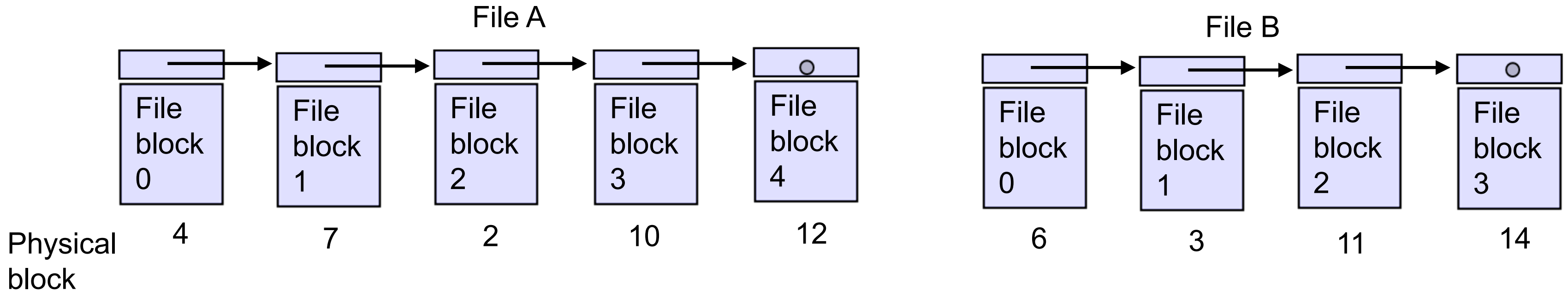
- Contiguous allocation
  - Each file is a contiguous run of disk blocks
  - e.g. IBM VM/CMS
  - Pros:
    - Simple to implement
    - Excellent read performance
  - Cons:
    - Fragmentation

Where would it make sense?



Implementing files

- Linked list
  - Files as a linked list of blocks
  - Pros:
    - Every block gets used
    - Simple directory entry per file
  - Cons:
    - Random access is a pain
    - List info in block → block data size not a power of 2
    - Reliability (file kept together by pointers scattered throughout the disk)



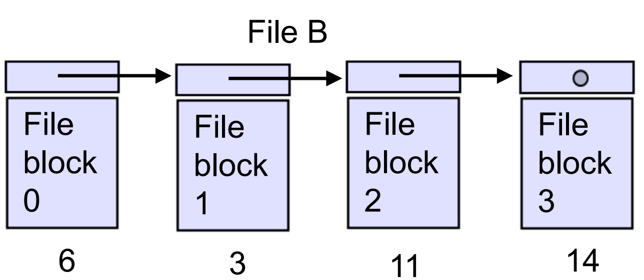
Implementing files

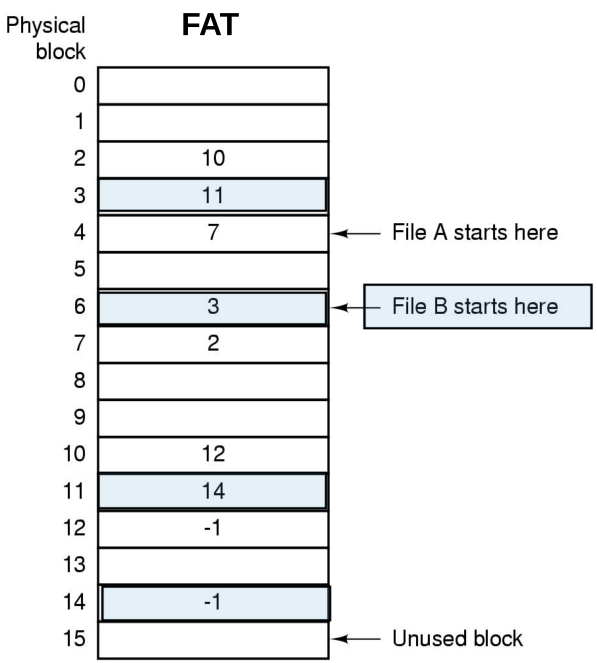
- Linked list with a table in memory
  - Files as a linked list of blocks
  - Pointers kept in FAT (File Allocation Table)
  - Pros:
    - Whole block free for data
    - Random access is easy
  - Cons:
    - Overhead on seeks or
    - Keep the entire table in memory

      20GB disk & 1KB block size →

      20 million entries in table →

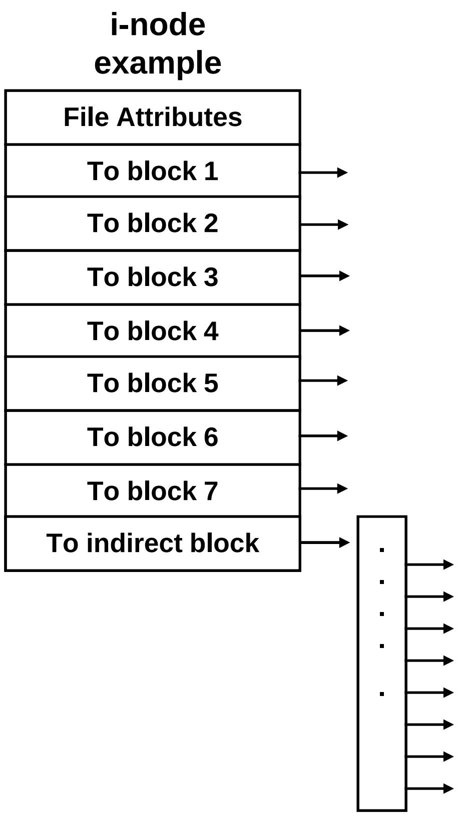
      4 bytes per entry ~ 80MB of memory





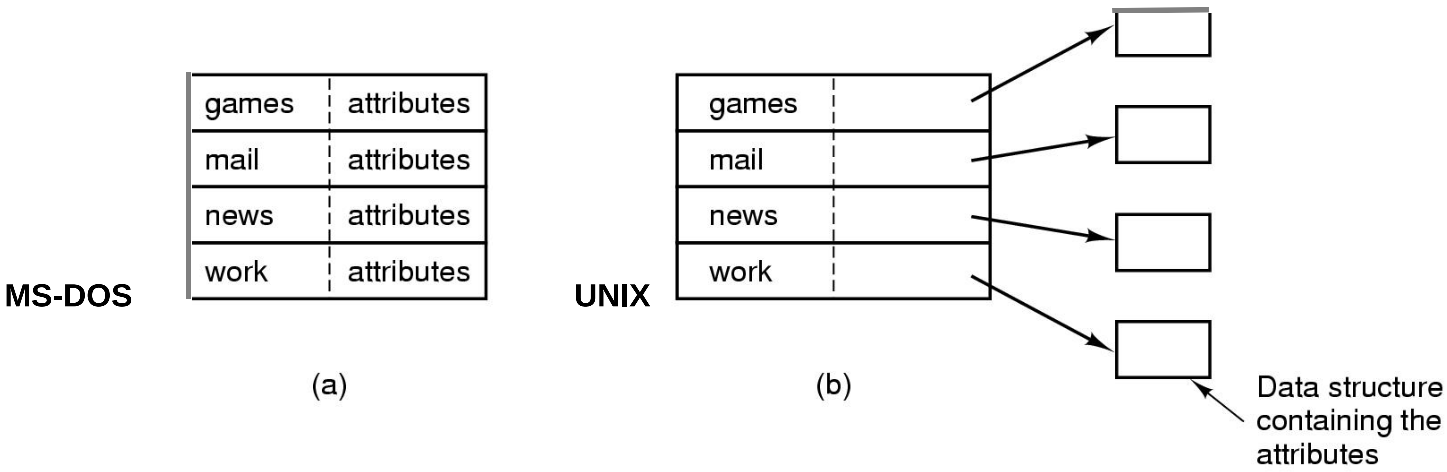
Implementing files

- I-nodes - index-nodes
  - Files as linked lists of blocks, all pointers in one location: i-node
  - Each file has its own i-node
  - Pros:
    - Support direct access
    - No external fragmentation
    - Only a file i-node needed in memory (proportional to # of open files instead of to disk size)
  - Cons:
    - Wasted space (how many entries?)
  - More entries? Save entry to point to address of block of addresses



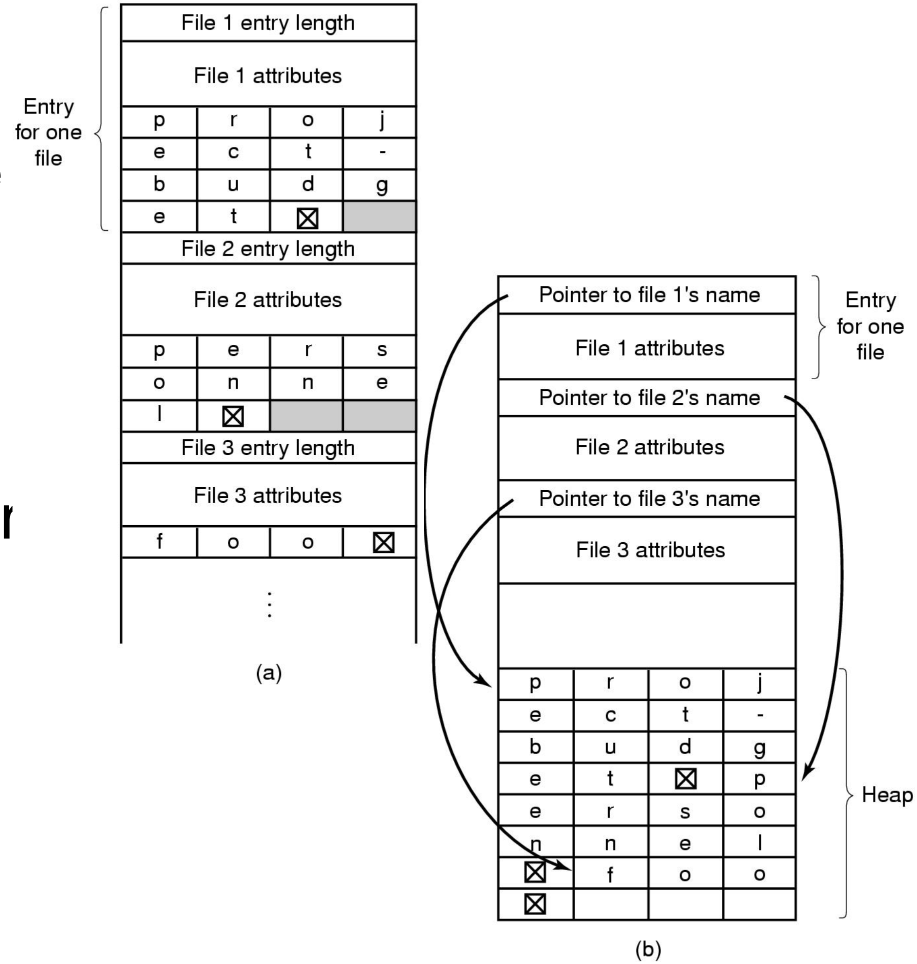
Implementing directories

- Directory system function: map ASCII name onto what's needed to locate the data
- Related: where do we store files' attributes?
  - A simple directory: fixed size entries, attributes in entry (a)
  - Directory in which each entry just refers to an i-node (b)
- As a side note, you find a file based on the path name; this mixes what your data is with where it is – *what's wrong with this picture?*



Implementing directories

- So far we've assumed short file names (8 or 14 char)
- Handling long file names in directory
  - In-line (a)
    - Fragmentation
    - Entry can span multiple pages (page fault reading a file name)
  - In a heap (b)
    - Easy to +/- files
- Searching large director
  - Hash
  - Cash



Shared files

- Links and directories implementation
  - Leave file's list of disk blocks out of directory entry (i-node)
    - Each entry in the directory points to the i-node
  - Use symbolic links
    - Link is a file w/ the path to shared file
    - Link files on another machine
- Problem with first solution
  - Accounting
    - C creates file, B links to file, C removes it
    - B is the only user of a file owned by C!
- Problem with symbolic links
  - Performance

Disk space management

- Once decided to store a file as sequence of blocks
  - What's the size of the block?
    - Good candidates: Sector, track, cylinder, page
    - Pros and cons of large/small blocks
    - Decide base on median file size (instead of mean)
- Keeping track of free blocks
  - Storing the free list on a linked list
    - Use a free block for the linked list
  - A bit map
- And if you run out of free space, control usage
  - Quotas for user's disk use
  - Open file entry includes pointer to owner's quota rec.
  - Soft limit may be exceeded (warning)
  - Hard limit may not (log in blocked)

File system reliability

- Need for backups
  - Bad things happen & while HW 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
- 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

File system reliability

- Logical dumps
  - Keep a bitmap indexed by i-node number
  - Bits are set for
    - Modified files
    - Directories
  - Unmarked directories w/o modified files in or under them
  - Dump directories and files marked
- Some more details
  - Free list is not dump, reconstructed
  - Unix files may have holes (core files are a good example)
  - Special files, named pipes, etc. are not dumped

File system reliability

- File system consistency
- fsck/scandisk ideas
  - Two kind of consistency checks: blocks & files
  - Blocks:
    - Build two tables – a counter per block and one pass
  - Similar check for directories – link counters kept in i-nodes

Consistent state

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Blocks in use	1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0
Free blocks	0	0	1	0	1	0	0	0	0	1	1	0	0	0	1	1

Missing block

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Blocks in use	1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0
Free blocks	0	0	0	0	1	0	0	0	0	1	1	0	0	0	1	1

Solution – add it to the free list

Twice in free list

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Blocks in use	1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0
Free blocks	0	0	1	0	2	0	0	0	0	1	1	0	0	0	1	1

Solution – rebuild the free list

Part of more than one file

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Blocks in use	1	1	0	1	0	2	1	1	1	0	0	1	1	1	0	0
Free blocks	0	0	1	0	1	0	0	0	0	1	1	0	0	0	1	1

Solution – duplicate data block

File system performance

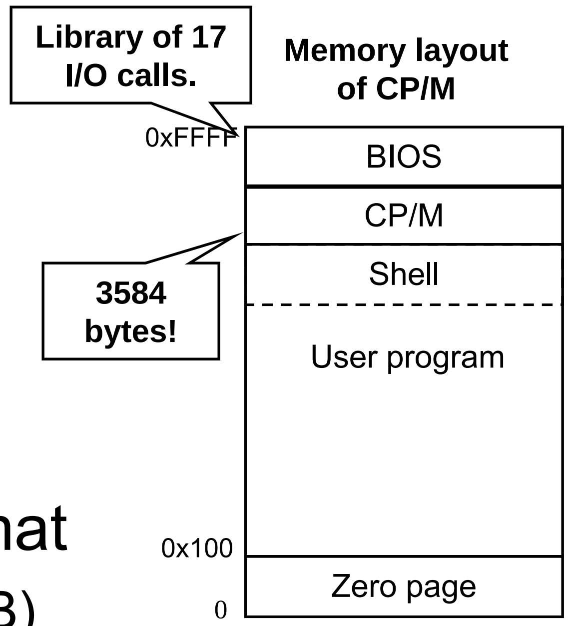
- Caching – to reduce disk access
  - Hash (device & disk address) to find block in cache
  - Cache management ~ page replacement
  - Plain LRU is undesirable
    - Essential blocks should be written out right away
    - If blocks would not be needed again, no point on caching
  - Unix sync and MS-DOS write-through cache
- Block read ahead
  - Clearly useless for non-sequentially read files
- Reducing disk arm motion
  - Put blocks likely to be accessed in seq. close to each other
  - I-nodes placed at the start of the disk
  - Disk divided into cylinder groups - each with its own blocks & i-nodes

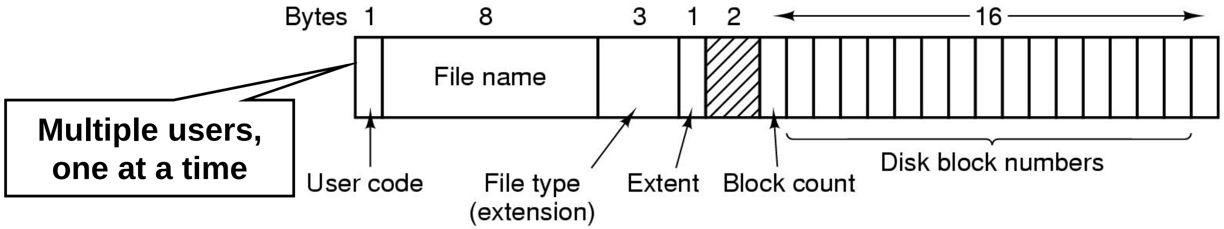
Log-structured file systems

- CPUs getting faster, memories larger, disks bigger
  - But disk seek time lags behind
  - Since disk caches can also be larger → increasing number of read requests can come from cache
  - *Thus, most disk accesses will be writes*
- LFS strategy - structure entire disk as a log
  - All writes initially buffered in memory
  - Periodically write buffer to end of disk log
    - Each new segment has a summary at the start
  - When file opened, locate i-node, then find blocks
    - Keep an i-node map in disk, index by i-node, and cache it
  - To deal with finite disks: cleaner thread
    - Compact segments starting at the front, first reading the summary, creating a new segment, marking the old one free

The CP/M file system

- Control Program for Microcomputers
- Run on Intel 8080 and Zilog Z80
  - 64KB main memory
  - 720KB floppy as secondary storage
- Separation bet/ 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

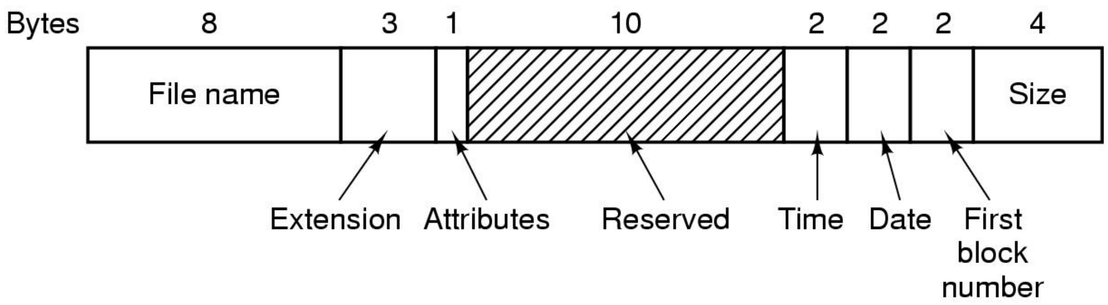




The MS-DOS file system

- Based on CP/M
- Biggest improvement: hierarchical file systems (v2.0)
  - Directories stored as files – no bound on hierarchy
  - No links – so basic tree
- Attributes include: read-only, hidden, archived, system
- Time – 5b for seconds, 6b for minutes, 5b for hours
  - Accurate only to +/-2 sec (2B – 65,536 sec of 86,400 sec/day)
- Date – 7b for year (128 years) starting at 1980 (5b for day, 4b for month)

MS-DOS directory entry

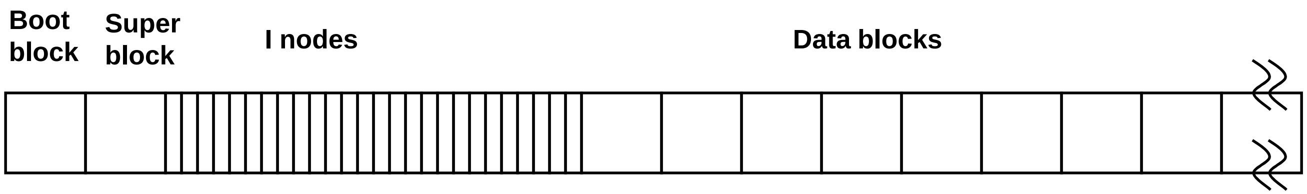


The MS-DOS file system

- Another difference with CP/M – FAT
  - First version FAT-12 with 512-byte blocks:
  - Max. partition 2^{12}x 512 ~ 2MB
  - FAT with 4096 entries of 2 bytes each – 8KB
- Later versions' FATs: FAT-16 and FAT-32 (actually a misnomer – only the low-order 28-bits are used)
- Disk block sizes can be set to multiple of 512B
- FAT-16:
  - 128KB of memory
  - Largest partition – 2GB ~ with block size 32KB
  - Largest disk - 8GB

The UNIX V7 file system

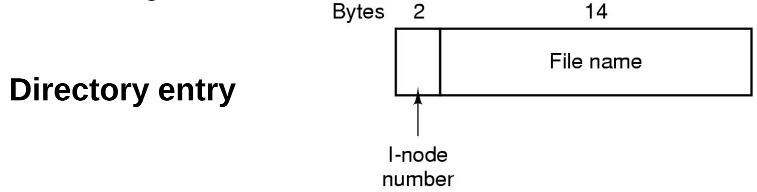
- Unix V7 on a PDP-11
- Tree structured as a DAG
- File names up to 14 chars (anything but “/” and NUL)
- Disk layout in classical UNIX systems



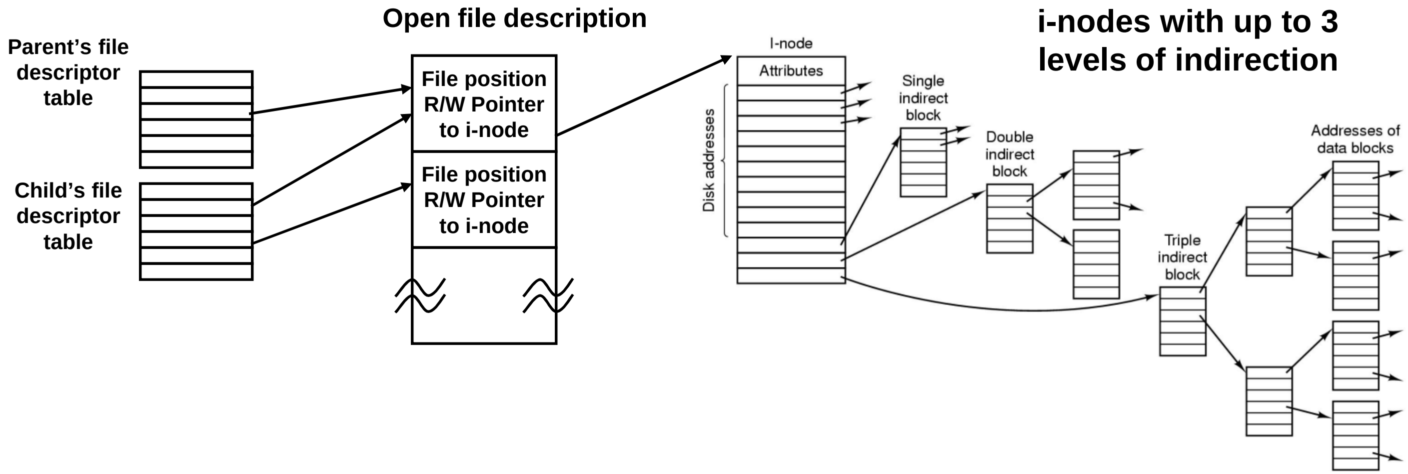
- Each i-node – 64 bytes long
- I-node’s attributes
  - file size, three times (creation, last access, last modif.), owner, group, protection info, # of dir entries pointing to it
- Following the i-nodes – data blocks in no particular order

The UNIX V7 file system

- A directory – an unsorted collection of 16-bytes entries

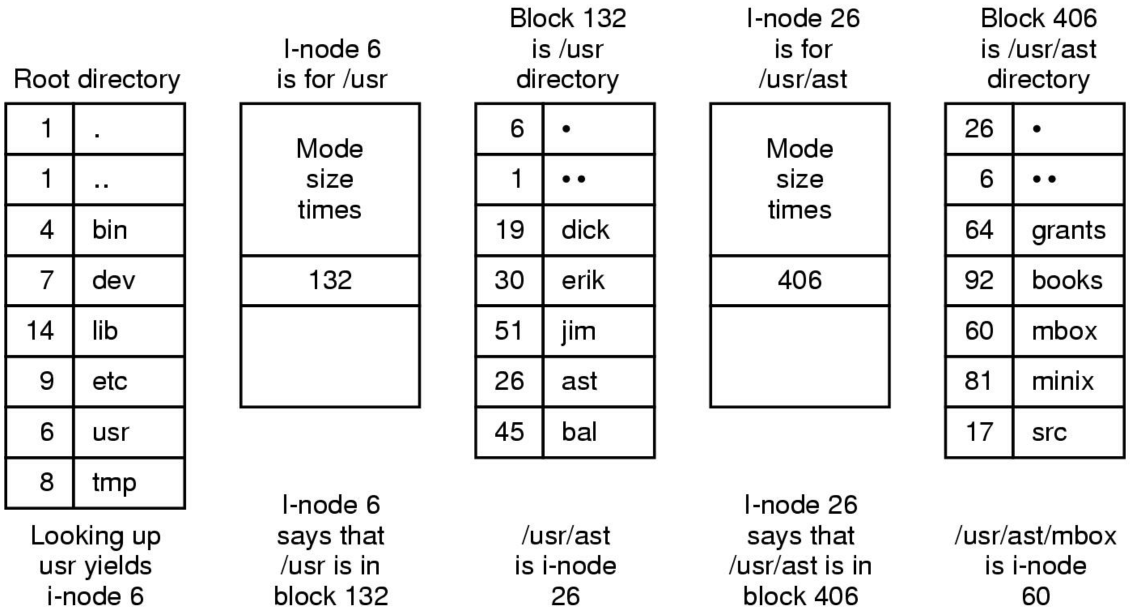


- File descriptor table, open file descriptor table and i-node table – starting from file descriptor, get the i-node
  - Pointer to i-node in the file descriptor table? No, where do you put the current pointer? Multiple processes each w/ their own
  - New table – the open file description



The UNIX V7 file system

- Steps in looking up /usr/ast/mbox
  – Locate root directory – i-node in a well-known place
  – Read root directory
  – Look for i-node for /usr
  – Read /usr and look for ast
  – …



Next Time

- Distributed systems
  - A quick introduction
- Research in OS
  - *READ the paper*
  - *I'll post a question in the course site the day before*
  - *Homework 5: your answer to this question; due in class*