Disk Allocation

ECE595
Mar 6

Y. Charlie Hu

[week8] Why Files?

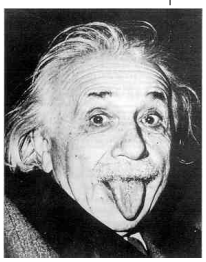
- Physical reality
  - Block oriented
  - Physical sector numbers
  - No protection among users of the system
  - Data might be corrupted if machine crashes
- File system abstraction
  - Byte oriented
  - Named files
  - Users protected from each other
  - Robust to machine failures

"I will save our lab3 solution on platter 5, track 8739, sector 3-4."

"My lab3 solution is in ~/ece595/lab3/memory.c"

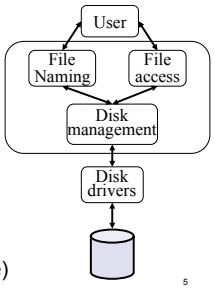
[week8] What Makes File Systems Hard?

- Files grow and shrink
  - Little *a priori* knowledge
  - 6 orders of magnitude in file sizes
- Unknown access patterns
- Overcoming disk performance behavior
  - Nonuniform acccess
  - Desire for efficiency
- Coping with failure



Roadmap

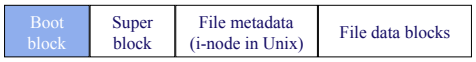
- Functionality (API)
  - → Basic file system
    - Data structures / disk layout
    - File operations
  - Directories
- Performance
  - Disk allocation
  - Disk scheduling
  - Buffer cache
    - interactions with VM
  - File system interface (alternative)
  - RAID



Definitions

- *File descriptor* (fd) – an integer used to represent a file – easier than using names
- *Metadata* – bookkeeping data that describes the file or info about it; not the actual content of file
  - *inode* – "index node", file metadata on Unix
- *Open file table* – system-wide list of file metadata in use

A Disk Layout for A File System



- Boot block: contains info to boot OS
- Superblock defines a file system
  - Size of the file system: numberf/size of blocks
  - Free metadata (inode) count and pointers
  - Free block count and pointers (or pointer to bitmap)
  - Location of the metadata of the root directory
- What if the superblock is corrupted?
  - What can we do?

Disk management: Data Structures (to keep track)

- Used space on disk:
  - A "header" for each file (part of the file meta-data)
    - Point to Disk blocks associated with each file
- Free space on disk
  - Bitmap
    - 1 bit per block (sector)
    - blocks numbered in cylinder-major order (why?)
  - Linked list of free blocks
  - How much space does a bitmap need for a 4GB disk?
    - 4294967296 bytes → 8388608 sectors → 1MB bitmap

File System API

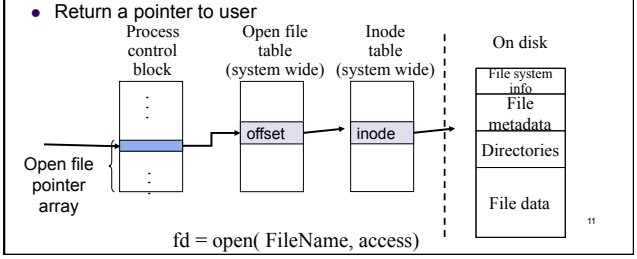
- OS provides the file system abstraction
- How do application processes access the file system?

File operations

- fd = open("/file1")
- read(fd, buf, size)
- write(fd, buf, size)
- close(fd)

Opening a File: fd = open("file1")

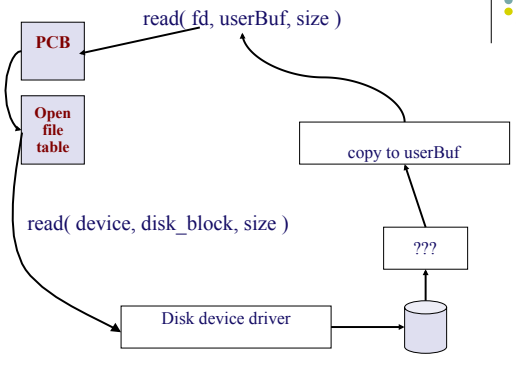
- File name lookup and authentication
- Create an entry in the open file table (system wide) if it is not in
- Copy the file metadata to in-memory data structure, if it is not in
- Create an entry in PCB
- Link up the data structures
- Return a pointer to user



[week1] Process Control Block (Process Table)

- Process management info
  - State (ready, running, blocked)
  - PC & Registers, parents, etc
  - CPU scheduling info (priorities, etc.)
- Memory management info
  - Segments, page table, stats, etc
- I/O and file management
  - Communication ports, directories, file descriptors, etc.

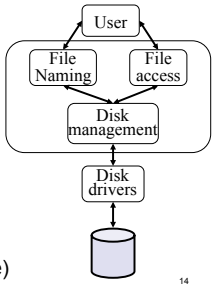
Reading a file (system call)



Modern disk drives are addressed as large one-dimensional arrays of logical blocks

Roadmap

- Functionality (API)
  - Basic file system
    - Data structures / disk layout
    - File operations
  - Directories
- Performance
  - → Disk allocation
  - Disk scheduling
  - Buffer cache
    - interactions with VM
  - File system interface (alternative)
  - RAID



Disk Allocation Problem

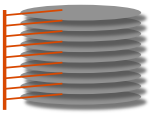
- Definition: allocate disk blocks when a file is created or grows, and free them when a file is removed or shrinks
- Does this sound familiar?
- Shall we approach it like segmentation or paging?
- Is there locality in disk accesses?
- What kind of locality matters?
  - Temporal?
  - Spatial?

Disk allocation problem

- Two tasks:
  - How to allocate blocks for a file?
  - How to design inode to keep track of blocks?

Disk mechanics & performance

- Platter / Head / Tracks / Sectors / Cylinders
- Rotation 1000's of RPM (7200, 10k, 15k)
- Avg seek 5-10 ms



- Assume
  - 255 heads *38913 tracks * 63 sectors * 512 bytes = 320GB
  - Seek time =6ms, 7200 RPM ➔ rotational latency = 8ms
- Block access time = seek time + rotational latency + reading time
  - Accessing a random block: 6ms + 4 ms + 8ms/63 = 638ms/63
  - Accessing the block right after: 8ms/63
- Implications?

Disk vs. Memory

Memory

- Latency in 100's of processor cycles
- Transfer rate ~1600 MB/s (DDR SDRAM) for 100MHz bus
- Contiguous alloc/ access gains ~10x
  - Cache hits
  - RAS/CAS (DRAM)

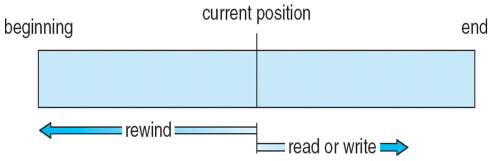
Disk

- Latency in milliseconds
  - 10ms = 10^7 cycles on 1Ghz machine
- Transfer rate in 30KB/s -- 30MB/s
- Contiguous alloc/ access gains 10~100x

Challenge to disk allocation problem: File Usage Patterns

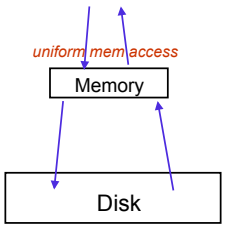
- How do users access files?
  - Sequential: bytes read in order
  - Random: read/write element out of middle of arrays
  - Whole file or partial file
- How are files used (determines metadata design)?
  - Most files are small
  - Large files use up most of the disk space
  - Large files account for most of the bytes transferred
- Bad news
  - Want everything to be efficient

Sequential File Access



[week7] Demand Paging algorithms

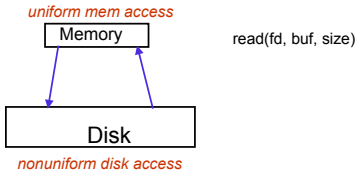
- Optimal
- FIFO
- FIFO with 2nd chance
- Clock: a simple FIFO with 2nd chance
- Enhanced FIFO with 2nd chance NFU
- Approximate LRU



- Definition: pick victim page to swap to disk upon page fault
- Focus on reducing number of page faults (going to disk)
  - Don't care which page fault, don't care where on disk
- What kind of *locality* matters?

Disk Allocation Problem

- Definition: allocate disk blocks when a file is created or grows, and free them when a file is removed or shrinks
- Cannot do anything about number of I/O -- Focus on performance of I/Os (that go to disk), i.e. which block
  - What kind of locality matters?



Design goal and expectation

- Optimize I/O performance
  - What can we do for random access patterns?
  - What can we do for sequential access patterns?
- Also want to minimize file header size
  - File header: for keeping track of blocks
  - Ideally fit in inode

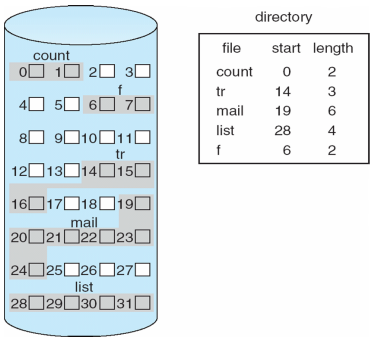
Disk Allocation Methods

- Contiguous
- Single-level indexed
- Linked
- FAT (MS-DOS, OS/2)
- Multi-level indexed (UNIX)

1. Contiguous Allocation

- Request in advance for the size of the file
- Search bit map or linked list to locate a space
- File header contains
  - first sector number
  - number of sectors

Contiguous Allocation of Disk Space



directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Analogy in memory management?

Contiguous Allocation

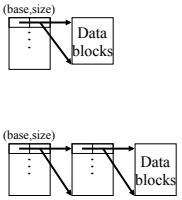
- Request in advance for the size of the file
- Search bit map or linked list to locate a space
- File header contains
  - first sector number
  - number of sectors
- Pros
  - Fast sequential access
  - Easy random access (how many I/Os for one I/O op)?
- Cons
  - External fragmentation
  - Hard to grow files

2. Extent-Based Systems

- Many newer file systems (i.e. Veritas File System – 1st commercial journal FS) use a modified contiguous allocation scheme
- Extent-based file systems allocate disk blocks in **extents**
- An **extent** is a contiguous block of disks
  - Extents are allocated for file allocation
  - A file consists of one or more extents

Example: DEMOS (OS for Cray-1, 1977)

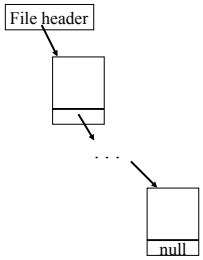
- Idea
  - Using contiguous allocation
  - Allow non-contiguous
- Approach
  - 10 (base,size) pointers
  - Indirect for big files
- Pros & cons
  - Can grow (max 10GB)
  - fragmentation
  - Difficult to grow each segment



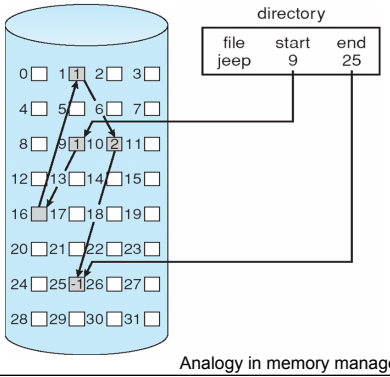
Analogy in memory management?

3. Linked Files

- File header points to 1st block on disk
- Each block points to next
- Pros
  - Can grow files dynamically
  - No external fragmentation
  - Sequential access easy
- Cons
  - random access: horrible
  - unreliable: losing a block means losing the rest



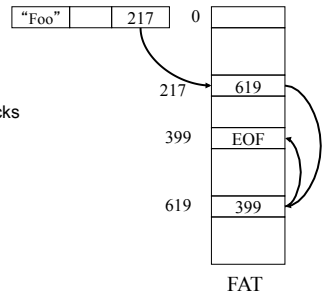
Linked Allocation



Analogy in memory management?

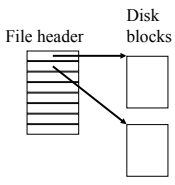
A variation of linked files: File Allocation Table (FAT) (MS-DOS, OS/2)

- Approach
  - A section of disk for each partition is reserved
  - One entry for each block
  - A file is a linked list of blocks
  - A directory entry points to the 1st block of the file
- Pros
  - Simpler than linked alloc.
- Cons
  - Always go to FAT
  - Random access slow for large files

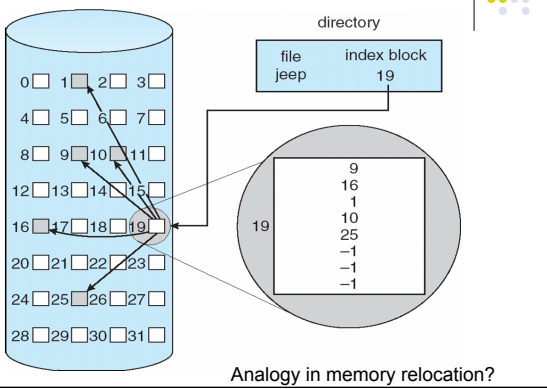


4. Single-Level Indexed Files

- A user declares max size
- A file header holds an array of pointers to point to disk blocks
- Pros
  - Can grow up to a limit
  - Random access is fast
  - No external fragmentation
- Cons
  - Clumsy to grow beyond limit
  - Up-front declaration a real pain
  - file header cannot fit in inode
    - large inodes would be wasteful



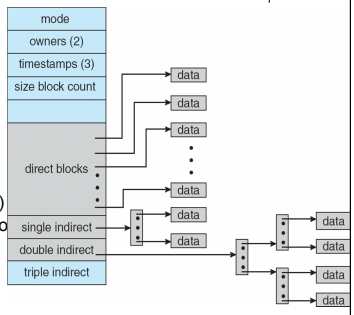
Example of Indexed Allocation



Analogy in memory relocation?

Multi-Level Indexed Files (UNIX)

- 13 Pointers in a header
  - 10 direct pointers
  - 11: 1-level indirect
  - 12: 2-level indirect
  - 13: 3-level indirect
- Pros & Cons
  - In favor of small files
  - Can grow
  - Limit is 16G (w/ 1K block)
  - Random access has up to 4 seeks
- How many disk accesses to load block 23, 5, 340?



Analogy in memory management?

Linux ext2

- From Linux kernel documentation for ext2:

  "*There are pointers to the first 12 blocks which contain the file's data in the inode. There is a pointer to an indirect block (which contains pointers to the next set of blocks), a pointer to a doubly indirect block and a pointer to a trebly indirect block.*"

- ext2 has 15 pointers.
  - Pointers 1 to 12 point to direct blocks
  - pointer 13 points to an indirect block
  - pointer 14 points to a doubly indirect block
  - pointer 15 points to a trebly indirect block

Theoretical ext2 limits under Linux

Block size:	1 KB	2 KB	4 KB	8 KB
max. file size:	16 GB	128 GB	1 TB	8 TB
max. filesystem size:	4 TB	8 TB	16 TB	32 TB

Deep thinking

- What about sequential access in multi-level indexed scheme?
- Can we try multi-level indexing in page table design?

Summary

- Seeks “kill” performance → exploit spatial locality
- Extent-based allocation optimizes sequential access
- Single-level indexed allocation has speed
- Unix file system has great flexibility
- Bitmaps show contiguous free space
- Linked lists easy to search for free blocks

Reading

- Chapters 10-11